engenharia de SOFTWARE

9ª edição

engenharia de
SOFTWARE

Ian
SOMMERVILLE

engenharia de SOFTWARE

9ª edição

Tradução
Kalinka Oliveira
Ivan Bosnic

Revisão Técnica
Prof. Dr. Kechi Hirama
Escola Politécnica da Universidade de São Paulo (EPUSP). Departamento de Engenharia de Computação e Sistemas Digitais (PCS). Laboratório de Tecnologia de Software (LTS). Grupo de Sistemas Complexos (GSC).

São Paulo

Brasil Argentina Colômbia Costa Rica Chile Espanha
Guatemala México Peru Porto Rico Venezuela

Tradução autorizada a partir da edição original, em inglês, *Software Engineering*, 9th edition, publicada pela Pearson Education, Inc., sob o selo Prentice Hall.

Diretor editorial: Roger Trimer
Gerente editorial: Sabrina Cairo
Editor de aquisição: Vinicius Souza
Coordenadora de produção editorial: Thelma Babaoka
Editora de texto: Aline Marques
Preparação: Renata Gonçalves
Revisão: Guilherme Summa e Camille Mendrot
Capa: Alexandre Mieda sobre projeto original de Elena Sidorova
Editoração eletrônica e diagramação: Figurativa Editorial MM Ltda.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Sommerville, Ian

Engenharia de Software / Ian Sommerville ; tradução Ivan Bosnic e Kalinka G. de O. Gonçalves ; revisão técnica Kechi Hirama. — 9. ed. — São Paulo : Pearson Prentice Hall, 2011.

Título original: Software engineering.

ISBN 978-85-7936-108-1

1. Engenharia de software I. Título.

11-02337 CDD-005.1

Índice para catálogo sistemático:
1. Engenharia de Software 005.1

2011
Direitos exclusivos para a língua portuguesa cedidos à
Pearson Education do Brasil,
uma empresa do grupo Pearson Education
Rua Nelson Francisco, 26, Limão
CEP: 02712–100 – São Paulo – SP
Tel: (11) 2178–8686 – Fax: (11) 2178–8688
e-mail: vendas@pearson.com

Sumário

Prefácio

Quando estava escrevendo os últimos capítulos deste livro, no verão de 2009, percebi que a engenharia de software havia completado 40 anos. O nome 'engenharia de software' foi proposto em 1969, na conferência da OTAN, para a discussão de problemas relacionados com desenvolvimento de software — grandes softwares atrasavam, não entregavam a funcionalidade de que os usuários necessitavam, custavam mais do que o esperado e não eram confiáveis. Eu não estava presente na conferência, porém, um ano depois, escrevi o meu primeiro programa e iniciei a minha carreira profissional em software.

O progresso da engenharia de software foi excepcional durante a minha carreira. Nossas sociedades não poderiam funcionar sem os grandes e profissionais sistemas de software. Para construir sistemas corporativos, existe uma sopa de letrinhas de tecnologias — J2EE, .NET, SaaS, SAP, BPEL4WS, SOAP, CBSE etc. — que suportam o desenvolvimento e a implantação de grandes aplicações corporativas. Os serviços e a infraestrutura nacionais — energia, comunicações e transporte — dependem de sistemas computacionais complexos e bastante confiáveis. O software nos permitiu explorar o espaço e criar a World Wide Web, o mais importante sistema de informação na história da humanidade. Enfrentamos agora um conjunto novo de desafios: mudança climática e tempo extremo, diminuição de recursos naturais, uma população mundial que não para de crescer e que precisa ser alimentada e abrigada, terrorismo internacional e a necessidade de ajudar pessoas mais idosas a terem uma vida mais satisfatória e com mais qualidade. Precisamos de novas tecnologias para nos ajudar a resolver esses problemas, e o software certamente terá um papel fundamental nessas tecnologias.

A engenharia de software é, portanto, uma tecnologia de importância crítica para o futuro da humanidade. Devemos continuar a educar engenheiros de software e a desenvolver a disciplina para podermos criar sistemas de software mais complexos. É claro que ainda há problemas com os projetos de software. Às vezes, o software ainda atrasa e custa mais do que o esperado. No entanto, não devemos deixar que esses problemas ofusquem os verdadeiros sucessos na engenharia de software e os métodos e as tecnologias impressionantes de engenharia de software que foram desenvolvidos.

A engenharia de software tornou-se uma área tão grande que é impossível cobrir todo o assunto em apenas um livro. Portanto, o meu foco estará em assuntos-chave que são fundamentais para todos os processos de desenvolvimento e temas que abordam o desenvolvimento de sistemas confiáveis e distribuídos. Há uma ênfase crescente em métodos ágeis e reúso de software. Eu, francamente, acredito que os métodos ágeis têm o seu lugar, mas também o tem a engenharia de software 'tradicional' dirigida a planejamento. Temos de combinar o melhor dessas abordagens para construir sistemas de software melhores.

Os livros refletem, inevitavelmente, as opiniões e os prejulgamentos dos seus autores. Do mesmo modo, alguns leitores certamente discordarão das minhas opiniões e da minha escolha de material. Tais discordâncias são um reflexo saudável da diversidade da disciplina e são essenciais para a sua evolução. Mesmo assim, espero que todos os engenheiros de software e estudantes da engenharia de software possam encontrar aqui algo de interessante.

Integração com a Internet

Há uma quantidade incrível de informações sobre a engenharia de software na Internet e algumas pessoas questionam se livros como este ainda são necessários. No entanto, a qualidade da informação disponível é bastan-

te questionável, a informação às vezes não é bem apresentada, e pode ser difícil encontrar aquilo que se procura. Consequentemente, eu acredito que os livros ainda têm, sim, um papel importante no ensino. Eles servem como um guia sobre o assunto e permitem que a informação sobre os métodos e as técnicas seja organizada e apresentada de forma coerente e de fácil leitura. Além disso, eles proveem um ponto de partida para uma exploração mais aprofundada da literatura de pesquisa e do material disponíveis na Internet.

Acredito plenamente que os livros têm um futuro, mas somente se forem integrados e agregarem valor à Internet. Por isso, este livro possui um Companion Website com materiais adicionais disponíveis 24 horas por dia. Veja a seção "Materiais de apoio" neste Prefácio.

Público-alvo

O livro se destina, em primeiro lugar, a estudantes de faculdades e universidades que estejam frequentando aulas introdutórias ou avançadas de engenharia de sistemas e de software. Os engenheiros de software no mercado de trabalho podem achar o livro útil como uma leitura geral e como um meio para atualizar os seus conhecimentos sobre assuntos como reúso de software, projeto de arquitetura, confiança e proteção e melhoria de processos. Suponho que o leitor tenha concluído um curso introdutório de programação e que esteja familiarizado com a terminologia de programação.

Mudanças em relação às edições anteriores

Esta edição conservou o material principal sobre a engenharia de software coberto em edições anteriores, porém eu revisei e atualizei todos os capítulos e acrescentei material novo em diversos pontos. As mudanças mais importantes são:

1. A reestruturação completa para tornar a obra mais fácil para se lecionar engenharia de software. O livro agora possui quatro, em vez de oito partes, e cada parte pode ser usada de forma independente ou em combinação com outras partes como uma base para o curso de engenharia de software. As quatro partes são uma introdução para engenharia de software, confiança e proteção, engenharia de software avançada e gerenciamento de engenharia de software.
2. Vários assuntos das edições anteriores são apresentados de forma mais concisa em um único capítulo, com material extra sendo movido para a Internet.
3. Atualizei e revisei o conteúdo em todos os capítulos. Estimo que entre 30% e 40% de todo o texto tenha sido totalmente reescrito.
4. Adicionei novos capítulos sobre desenvolvimento de software ágil e sistemas embutidos.
5. Assim como esses novos capítulos, há material novo sobre engenharia dirigida a modelos, desenvolvimento *open source*, desenvolvimento dirigido a testes, modelo Swiss Cheese de Reason, arquiteturas de sistemas confiáveis, análise estática e verificação de modelos, reúso de COTS, software como um serviço e planejamento ágil.
6. Um novo estudo de caso sobre um sistema de registro de pacientes que necessitam de tratamento para problemas de saúde mental foi usado em vários capítulos.

Usando o livro para lecionar

Organizei o livro de tal forma que possa ser usado em três tipos diferentes de cursos de engenharia de software:

1. *Cursos gerais de introdução à engenharia de software.* A primeira parte do livro foi organizada especialmente para apoiar um curso de um semestre sobre introdução à engenharia de software.
2. *Cursos introdutórios ou intermediários sobre assuntos específicos da engenharia de software.* Você pode criar uma série de cursos mais avançados usando os capítulos das partes 2 até 4. Por exemplo, eu lecionei um curso sobre engenharia de sistemas críticos usando os capítulos da Parte 2, junto com os capítulos de gerenciamento de qualidade e gerenciamento de configuração.

3. *Cursos mais avançados sobre assuntos mais específicos da engenharia de software.* Nesse caso, os capítulos do livro formam a base para o curso. Estes são, depois, acrescidos de leituras mais aprofundadas que exploram o assunto em mais detalhes. Por exemplo, um curso sobre o reúso de software poderia ser baseado nos capítulos 16, 17, 18 e 19.

Materiais de apoio

No Companion Website deste livro www.pearson.com.br/sommerville, professores e estudantes podem acessar materiais adicionais 24 horas por dia. Estão disponíveis:

Para professores:

- Apresentações em PowerPoint.
- Sugestões de como utilizar o livro em sala de aula (em inglês).
- Banco de exercícios (em inglês).
- Manual de soluções (em inglês).

Esse material é de uso exclusivo dos professores e está protegido por senha. Para ter acesso a eles, os professores que adotam o livro devem entrar em contato com seu representante Pearson ou enviar e-mail para universitarios@pearson.com.

Para estudantes:

- Estudos de caso — fornecem informações adicionais sobre os estudos de caso usados no livro (bomba de insulina, sistema de saúde mental, sistema meteorológico remoto), assim como informações sobre outros estudos de caso, como a falha do lançador Ariane 5.
- Capítulos adicionais — existem quatro capítulos adicionais que cobrem métodos formais, projetos de interação, documentação e arquiteturas de aplicações.

Agradecimentos

Um grande número de pessoas contribuiu durante anos por a evolução deste livro e eu gostaria de agradecer a todos (revisores, estudantes e usuários do livro) que comentaram as edições anteriores e fizeram sugestões construtivas para as mudanças.

Gostaria de agradecer particularmente à minha família (Anne, Ali e Jane) por sua ajuda e apoio enquanto o livro estava sendo escrito. Um obrigado especial para a minha filha Jane, que descobriu um talento para revisão e edição. Ela foi excepcionalmente útil ao ler o livro inteiro e fez um ótimo trabalho detectando e corrigindo um número grande de erros gramaticais e de digitação.

Ian Sommerville

PARTE

1 2 3 4

Introdução à engenharia de software

O objetivo nesta parte do livro é fornecer uma introdução geral à engenharia de software. Apresento conceitos importantes, como processos de software e métodos ágeis, e descrevo as atividades essenciais para o desenvolvimento de software, desde a especificação inicial do software até a evolução do sistema. Nesta primeira parte, os capítulos foram concebidos para darem suporte a um curso de engenharia de software de um semestre.

O Capítulo 1 é uma apresentação geral que apresenta a engenharia de software profissional e define alguns conceitos da área. Também escrevi uma breve discussão sobre as questões éticas na engenharia de software. Acho que é importante os engenheiros de software pensarem sobre as implicações mais amplas do seu trabalho. Este capítulo também apresenta três estudos de caso, um sistema de gerenciamento de registros de pacientes em tratamento para problemas de saúde mental, um sistema de controle para uma bomba de insulina portátil e um sistema meteorológico no deserto.

Os capítulos 2 e 3 abrangem os processos de engenharia de software e desenvolvimento ágil. No Capítulo 2, apresento modelos genéricos de processos de software, como o modelo em cascata, e discuto as atividades básicas que são parte desses processos. O Capitulo 3 suplementa esse, com uma discussão sobre métodos ágeis de desenvolvimento de engenharia de software. Uso ainda a Extreme Programming como exemplo de método ágil, mas neste capítulo também faço uma leve introdução ao Scrum.

O restante dos capítulos desta parte são descrições detalhadas das atividades de processo de software, as quais serão introduzidas no Capitulo 2. O Capítulo 4 aborda o tema crítico de importância de engenharia de requisitos, em que são definidos os requisitos que especificam o que um sistema deve fazer. O Capítulo 5 apresenta a modelagem de sistemas usando a UML centrada no uso de diagramas de caso de uso, diagramas de classe, diagramas de sequência e diagramas de estado para a modelagem de um sistema de software. O Capítulo 6 apresenta os projetos de arquitetura, em que se discute a importância da arquitetura e do uso de padrões em projetos de software.

O Capítulo 7 apresenta o projeto orientado a objetos e o uso de padrões de projeto. Apresento também importantes questões de implementação — reuso, gerenciamento de configuração e desenvolvimento *host-target*, além de discutir o desenvolvimento *open source*. O Capítulo 8 se concentra nos testes de software desde os testes unitários durante o desenvolvimento do sistema até o teste de releases de software. Discuto ainda o uso do desenvolvimento dirigido a testes — de uma perspectiva pioneira em métodos ágeis, mas de grande aplicabilidade. Finalmente, o Capítulo 9 apresenta uma visão geral dos assuntos relacionados à evolução de software. Abrange os processos de evolução e manutenção de software, e gerenciamento de sistemas legados.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Introdução

Objetivos

Os objetivos deste capítulo são fazer uma introdução à engenharia de software e fornecer uma base para a compreensão do restante do livro. Depois de ler este capítulo, você:

- entenderá o que é engenharia de software e por que ela é importante;
- entenderá que o desenvolvimento de diferentes tipos de sistemas de software pode requerer diferentes técnicas de engenharia de software;
- entenderá algumas questões éticas e profissionais importantes para engenheiros de software;
- terá conhecimento de três sistemas de tipos diferentes que serão usados como exemplos neste livro.

Conteúdo

O mundo moderno não poderia existir sem o software. Infraestruturas e serviços nacionais são controlados por sistemas computacionais, e a maioria dos produtos elétricos inclui um computador e um software que o controla. A manufatura e a distribuição industriais são totalmente informatizadas, assim como o sistema financeiro. A área de entretenimento, incluindo a indústria da música, jogos de computador, cinema e televisão, faz uso intensivo de software. Portanto, a engenharia de software é essencial para o funcionamento de sociedades nacionais e internacionais.

Os sistemas de software são abstratos e intangíveis. Eles não são restringidos pelas propriedades dos materiais, nem governados pelas leis da física ou pelos processos de manufatura. Isso simplifica a engenharia de software, porque não há limites naturais para o potencial do software. No entanto, devido a essa falta de restrições físicas, os sistemas de software podem se tornar extremamente complexos de modo muito rápido, difíceis de entender e caros para alterar.

Existem vários tipos de sistemas de software, desde os simples sistemas embutidos até os sistemas de informações complexos, de alcance mundial. Não faz sentido procurar notações, métodos ou técnicas universais para a engenharia de software, porque diferentes tipos de software exigem abordagens diferentes. Desenvolver um sistema de informações corporativo é totalmente diferente de desenvolver um controlador para um instrumento científico. Nenhum desses sistemas tem muito em comum com um jogo computacional com gráficos intensos. Todas essas aplicações precisam de engenharia de software, embora não necessitem das mesmas técnicas.

Ainda existem muitos relatos e projetos de software que deram errado e resultaram em 'falhas de software'. A engenharia de software é criticada por ser inadequada para o desenvolvimento moderno de software. No entanto, no meu ponto de vista, muitas dessas falhas são consequência de dois fatores:

1. *Aumento de demanda.* Conforme novas técnicas de engenharia de software nos auxiliam a construir sistemas maiores e mais complexos, as demandas mudam. Os sistemas têm de ser construídos e entregues mais rapidamente;

sistemas maiores e até mais complexos são requeridos; sistemas devem ter novas capacidades que antes eram consideradas impossíveis. Como os métodos de engenharia de software existentes não conseguem lidar com isso, novas técnicas de engenharia de software precisam ser desenvolvidas para atender a essas novas demandas.

2. *Expectativas baixas*. É relativamente fácil escrever programas computacionais sem usar técnicas e métodos de engenharia de software. Muitas empresas foram forçadas a desenvolver softwares à medida que seus produtos e serviços evoluíram. Elas não usam métodos de engenharia de software no dia a dia. Consequentemente, seu software é frequentemente mais caro e menos confiável do que deveria ser. Precisamos de educação e treinamento em engenharia de software para solucionar esses problemas.

Engenheiros de software têm o direito de se orgulhar de suas conquistas. É claro que ainda temos problemas em desenvolver softwares complexos, mas, sem a engenharia de software, não teríamos explorado o espaço, não teríamos a Internet ou as telecomunicações modernas. Todas as formas de viagem seriam mais perigosas e caras. A engenharia de software contribuiu muito, e tenho certeza de que suas contribuições no século XXI serão maiores ainda.

1.1 Desenvolvimento profissional de software

Inúmeras pessoas escrevem programas. Pessoas envolvidas com negócios escrevem programas em planilhas para simplificar seu trabalho; cientistas e engenheiros escrevem programas para processar seus dados experimentais; e há aqueles que escrevem programas como *hobby*, para seu próprio interesse e diversão. No entanto, a maior parte do desenvolvimento de software é uma atividade profissional, em que o software é desenvolvido para um propósito específico de negócio, para inclusão em outros dispositivos ou como produtos de software como sistemas de informação, sistemas CAD etc. O software profissional, o que é usado por alguém além do seu desenvolvedor, é normalmente criado por equipes, em vez de indivíduos. Ele é mantido e alterado durante sua vida.

A engenharia de software tem por objetivo apoiar o desenvolvimento profissional de software, mais do que a programação individual. Ela inclui técnicas que apoiam especificação, projeto e evolução de programas, que normalmente não são relevantes para o desenvolvimento de software pessoal. Para ajudá-lo a ter uma visão geral sobre o que trata a engenharia de software, listei algumas perguntas comuns na Tabela 1.1.

Muitas pessoas pensam que software é simplesmente outra palavra para programas de computador. No entanto, quando falamos de engenharia de software, não se trata apenas do programa em si, mas de toda a documentação associada e dados de configurações necessários para fazer esse programa operar corretamente. Um sistema de software desenvolvido profissionalmente é, com frequência, mais do que apenas um programa; ele normalmente consiste em uma série de programas separados e arquivos de configuração que são usados para configurar esses programas. Isso pode incluir documentação do sistema, que descreve a sua estrutura; documentação do usuário, que explica como usar o sistema; e sites, para usuários baixarem a informação recente do produto.

Essa é uma diferença importante entre desenvolvimento de software profissional e amador. Se você está escrevendo um programa para si mesmo, que ninguém mais usará, você não precisa se preocupar em escrever o manual do programa, documentar sua arquitetura etc. No entanto, se você está escrevendo um software que outras pessoas usarão e no qual outros engenheiros farão alterações, então você provavelmente deve fornecer informação adicional, assim como o código do programa.

Engenheiros de software se preocupam em desenvolver produtos de software (ou seja, software que pode ser vendido para um cliente). Existem dos tipos de produtos de software:

1. *Produtos genéricos*. Existem sistemas *stand-alone*, produzidos por uma organização de desenvolvimento e vendidos no mercado para qualquer cliente que esteja interessado em comprá-los. Exemplos desse tipo de produto incluem software para PCs, como ferramentas de banco de dados, processadores de texto, pacotes gráficos e gerenciamento de projetos. Também incluem as chamadas aplicações verticais projetadas para um propósito específico, como sistemas de informação de bibliotecas, sistemas de contabilidade ou sistemas de manutenção de registros odontológicos.
2. *Produtos sob encomenda*. Estes são os sistemas encomendados por um cliente em particular. Uma empresa de software desenvolve o software especialmente para esse cliente. Exemplos desse tipo de software são sistemas de controle de dispositivos eletrônicos, sistemas escritos para apoiar um processo de negócio específico e sistemas de controle de tráfego aéreo.

Tabela 1.1 Perguntas frequentes sobre software

Pergunta	Resposta
O que é software?	Softwares são programas de computador e documentação associada. Produtos de software podem ser desenvolvidos para um cliente específico ou para o mercado em geral.
Quais são os atributos de um bom software?	Um bom software deve prover a funcionalidade e o desempenho requeridos pelo usuário; além disso, deve ser confiável e fácil de manter e usar.
O que é engenharia de software?	É uma disciplina de engenharia que se preocupa com todos os aspectos de produção de software.
Quais são as principais atividades da engenharia de software?	Especificação de software, desenvolvimento de software, validação de software e evolução de software.
Qual a diferença entre engenharia de software e ciência da computação?	Ciência da computação foca a teoria e os fundamentos; engenharia de software preocupa-se com o lado prático do desenvolvimento e entrega de softwares úteis.
Qual a diferença entre engenharia de software e engenharia de sistemas?	Engenharia de sistemas se preocupa com todos os aspectos do desenvolvimento de sistemas computacionais, incluindo engenharia de hardware, software e processo. Engenharia de software é uma parte específica desse processo mais genérico.
Quais são os principais desafios da engenharia de software?	Lidar com o aumento de diversidade, demandas pela diminuição do tempo para entrega e desenvolvimento de software confiável.
Quais são os custos da engenharia de software?	Aproximadamente 60% dos custos de software são de desenvolvimento; 40% são custos de testes. Para software customizado, os custos de evolução frequentemente superam os custos de desenvolvimento.
Quais são as melhores técnicas e métodos da engenharia de software?	Enquanto todos os projetos de software devem ser gerenciados e desenvolvidos profissionalmente, técnicas diferentes são adequadas para tipos de sistemas diferentes. Por exemplo, jogos devem ser sempre desenvolvidos usando uma série de protótipos, enquanto sistemas de controle críticos de segurança requerem uma especificação analisável e completa. Portanto, não se pode dizer que um método é melhor que outro.
Quais diferenças foram feitas pela Internet na engenharia de software?	A Internet tornou serviços de software disponíveis e possibilitou o desenvolvimento de sistemas altamente distribuídos baseados em serviços. O desenvolvimento de sistemas baseados em Web gerou importantes avanços nas linguagens de programação e reúso de software.

Uma diferença importante entre esses tipos de software é que, em softwares genéricos, a organização que o desenvolve controla sua especificação. Para produtos sob encomenda, a especificação é normalmente desenvolvida e controlada pela empresa que está adquirindo o software. Os desenvolvedores de software devem trabalhar de acordo com essa especificação.

No entanto, a distinção entre esses tipos de produtos de software está se tornando cada vez mais obscura. Mais e mais sistemas vêm sendo construídos tendo por base um produto genérico, que é então adaptado para atender aos requisitos de um cliente. Sistemas ERP (sistema integrado de gestão empresarial, do inglês *enterprise resourse planning*), como o sistema SAP, são os melhores exemplos dessa abordagem. Nesse caso, um sistema grande e complexo é adaptado para uma empresa, incorporando informações sobre as regras e os processos de negócio, relatórios necessários etc.

Quando falamos sobre a qualidade do software profissional, devemos levar em conta que o software é usado e alterado pelas pessoas, além de seus desenvolvedores. A qualidade, portanto, implica não apenas o que o software faz. Ao contrário, ela tem de incluir o comportamento do software enquanto ele está executando, bem como a estrutura e a organização dos programas do sistema e a documentação associada. Isso se reflete nos atributos de software chamados não funcionais ou de qualidade. Exemplos desses atributos são o tempo de resposta do software a uma consulta do usuário e a compreensão do código do programa.

Um conjunto específico de atributos que você pode esperar de um software obviamente depende da aplicação. Portanto, um sistema bancário deve ser seguro, um jogo interativo deve ser ágil, um sistema de comutação de telefonia deve ser confiável, e assim por diante. Tudo isso pode ser generalizado em um conjunto de atributos sumarizados na Tabela 1.2, que eu acredito serem as características essenciais de um sistema profissional de software.

Tabela 1.2 Atributos essenciais de um bom software

Características do produto	Descrição
Manutenibilidade	O software deve ser escrito de forma que possa evoluir para atender às necessidades dos clientes. Esse é um atributo crítico, porque a mudança de software é um requisito inevitável de um ambiente de negócio em mudança.
Confiança e proteção	A confiança do software inclui uma série de características como confiabilidade, proteção e segurança. Um software confiável não deve causar prejuízos físicos ou econômicos no caso de falha de sistema. Usuários maliciosos não devem ser capazes de acessar ou prejudicar o sistema.
Eficiência	O software não deve desperdiçar os recursos do sistema, como memória e ciclos do processador. Portanto, eficiência inclui capacidade de resposta, tempo de processamento, uso de memória etc.
Aceitabilidade	O software deve ser aceitável para o tipo de usuário para o qual foi projetado. Isso significa que deve ser compreensível, usável e compatível com outros sistemas usados por ele.

1.1.1 Engenharia de software

Engenharia de software é uma disciplina de engenharia cujo foco está em todos os aspectos da produção de software, desde os estágios iniciais da especificação do sistema até sua manutenção, quando o sistema já está sendo usado. Há duas expressões importantes nessa definição:

1. *Disciplina de engenharia*. Engenheiros fazem as coisas funcionarem. Eles aplicam teorias, métodos e ferramentas onde for apropriado. No entanto, eles os usam seletivamente e sempre tentam descobrir as soluções para os problemas, mesmo quando não há teorias e métodos aplicáveis. Os engenheiros também reconhecem que devem trabalhar de acordo com as restrições organizacionais e financeiras, então buscam soluções dentro dessas restrições.
2. *Todos os aspectos da produção de software*. A engenharia de software não se preocupa apenas com os processos técnicos do desenvolvimento de software. Ela também inclui atividades como gerenciamento de projeto de software e desenvolvimento de ferramentas, métodos e teorias para apoiar a produção de software.

Engenharia tem a ver com obter resultados de qualidade requeridos dentro do cronograma e do orçamento. Isso frequentemente envolve ter compromissos — engenheiros não podem ser perfeccionistas. Por outro lado, as pessoas que escrevem programas para si mesmas podem gastar o tempo que quiserem com o desenvolvimento do programa.

Em geral, os engenheiros de software adotam uma abordagem sistemática e organizada para seu trabalho, pois essa costuma ser a maneira mais eficiente de produzir software de alta qualidade. No entanto, engenharia tem tudo a ver com selecionar o método mais adequado para um conjunto de circunstâncias, então uma abordagem mais criativa e menos formal pode ser eficiente em algumas circunstâncias. Desenvolvimento menos formal é particularmente adequado para o desenvolvimento de sistemas Web, que requerem uma mistura de habilidades de software e de projeto.

Engenharia de software é importante por dois motivos:

1. Cada vez mais, indivíduos e sociedades dependem dos sistemas de software avançados. Temos de ser capazes de produzir sistemas confiáveis econômica e rapidamente.
2. Geralmente é mais barato, a longo prazo, usar métodos e técnicas da engenharia de software para sistemas de software, em vez de simplesmente escrever os programas como se fossem algum projeto pessoal. Para a maioria dos sistemas, a maior parte do custo é mudar o software depois que ele começa a ser usado.

A abordagem sistemática usada na engenharia de software é, às vezes, chamada processo de software. Um processo de software é uma sequência de atividades que leva à produção de um produto de software. Existem quatro atividades fundamentais comuns a todos os processos de software. São elas:

1. Especificação de software, em que clientes e engenheiros definem o software a ser produzido e as restrições de sua operação.
2. Desenvolvimento de software, em que o software é projetado e programado.

3. Validação de software, em que o software é verificado para garantir que é o que o cliente quer.
4. Evolução de software, em que o software é modificado para refletir a mudança de requisitos do cliente e do mercado.

Tipos diferentes de sistemas necessitam de diferentes processos de desenvolvimento. Por exemplo, um software de tempo real em uma aeronave precisa ser completamente especificado antes de se iniciar o desenvolvimento. Em sistemas de comércio eletrônico, a especificação e o programa são, normalmente, desenvolvidos juntos. Consequentemente, essas atividades genéricas podem ser organizadas de formas diferentes e descritas em nível de detalhamento diferente, dependendo do tipo de software em desenvolvimento. Descrevo processos de software em mais detalhes no Capítulo 2.

Engenharia de software se relaciona tanto com ciência da computação quanto com engenharia de sistemas:

1. A ciência da computação se preocupa com as teorias e métodos que sustentam sistemas computacionais e de software, ao passo que a engenharia de software se preocupa com os problemas práticos de produção de software. Algum conhecimento de ciência da computação é essencial para engenheiros de software, da mesma forma que algum conhecimento de física é essencial para engenheiros elétricos. No entanto, a teoria da ciência da computação é, em geral, mais aplicável para programas relativamente pequenos. Teorias elegantes da ciência da computação nem sempre podem ser aplicadas em problemas grandes e complexos que requerem uma solução através de software.
2. A engenharia de sistemas foca todos os aspectos do desenvolvimento e da evolução de sistemas complexos em que o software tem o papel principal. A engenharia de sistemas se preocupa com desenvolvimento de hardware, projeto de políticas e processos e implantação de sistemas, além de engenharia de software. Engenheiros de sistemas são envolvidos em especificação do sistema, definição da arquitetura geral e integração de diferentes partes para criar o sistema acabado. Eles se preocupam menos com a engenharia dos componentes do sistema (hardware, software etc.).

Conforme discutido na próxima seção, existem muitos tipos de software. Não existe um método ou uma técnica universal de engenharia de software que se aplique a todos. No entanto, há três aspectos gerais que afetam vários tipos diferentes de software:

1. *Heterogeneidade*. Cada vez mais se requer dos sistemas que operem como sistemas distribuídos através das redes que incluem diferentes tipos de computadores e dispositivos móveis. Além de executar nos computadores de propósito geral, o software talvez tenha de executar em telefones móveis. Frequentemente, você tem de integrar software novo com sistemas mais antigos, escritos em linguagens de programação diferentes. O desafio aqui é desenvolver técnicas para construir um software confiável que seja flexível o suficiente para lidar com essa heterogeneidade.
2. *Mudança de negócio e social*. Negócio e sociedade estão mudando de maneira incrivelmente rápida, à medida que as economias emergentes se desenvolvem e as novas tecnologias se tornam disponíveis. Deve ser possível alterar seu software existente e desenvolver um novo software rapidamente. Muitas técnicas tradicionais de engenharia de software consomem tempo, e a entrega de novos sistemas frequentemente é mais demorada do que o planejado. É preciso evoluir para que o tempo requerido para o software dar retorno a seus clientes seja reduzido.
3. *Segurança e confiança*. Pelo fato de o software estar presente em todos os aspectos de nossas vidas, é essencial que possamos confiar nele. Isso se torna verdade especialmente para sistemas remotos acessados através de uma página Web ou uma interface de *web service*. Precisamos ter certeza de que os usuários maliciosos não possam atacar nosso software e de que a proteção da informação seja mantida.

É claro que essas questões não são independentes. Por exemplo, pode ser necessário fazer mudanças rápidas em um sistema legado para que se possa oferecer o mesmo com uma interface de *web service*. Para resolver esses desafios precisaremos de novas ferramentas e técnicas, bem como de maneiras inovadoras de combinar e usar métodos de engenharia de software existentes.

1.1.2 Diversidade na engenharia de software

Engenharia de software é uma abordagem sistemática para a produção de software; ela analisa questões práticas de custo, prazo e confiança, assim como as necessidades dos clientes e produtores do software. A forma como essa abordagem sistemática é realmente implementada varia dramaticamente de acordo com a organização que

esteja desenvolvendo o software, o tipo de software e as pessoas envolvidas no processo de desenvolvimento. Não existem técnicas e métodos universais na engenharia de software adequados a todos os sistemas e todas as empresas. Em vez disso, um conjunto diverso de métodos e ferramentas de engenharia de software tem evoluído nos últimos 50 anos.

Talvez o fator mais significante em determinar quais técnicas e métodos de engenharia de software são mais importantes seja o tipo de aplicação a ser desenvolvida. Existem muitos tipos diferentes de aplicações, incluindo:

1. *Aplicações* stand-alone. Essas são as aplicações executadas em um computador local, como um PC. Elas contêm toda a funcionalidade necessária e não precisam estar conectadas a uma rede. Exemplos de tais aplicações são aplicativos de escritório em um PC, programas CAD, software de manipulação de fotos etc.
2. *Aplicações interativas baseadas em transações*. São aplicações que executam em um computador remoto, acessadas pelos usuários a partir de seus computadores ou terminais. Certamente, aqui são incluídas aplicações Web como aplicações de comércio eletrônico em que você pode interagir com o sistema remoto para comprar produtos ou serviços. Essa classe de aplicações também inclui sistemas corporativos, em que uma empresa fornece acesso a seus sistemas através de um navegador Web ou um programa cliente especial e serviços baseados em nuvem, como é o caso de serviços de e-mail e compartilhamento de fotos. Aplicações interativas frequentemente incorporam um grande armazenamento de dados, que é acessado e atualizado em cada transação.
3. *Sistemas de controle embutidos*. São sistemas de controle que controlam e gerenciam dispositivos de hardware. Numericamente, é provável que haja mais sistemas embutidos do que de qualquer outro tipo. Exemplos de sistemas embutidos incluem software em telefone celular, softwares que controlam antitravamento de freios em um carro e software em um micro-ondas para controlar o processo de cozimento.
4. *Sistemas de processamento de lotes*. São sistemas corporativos projetados para processar dados em grandes lotes. Eles processam grande número de entradas individuais para criar as saídas correspondentes. Exemplos de sistemas de lotes incluem sistemas periódicos de cobrança, como sistemas de cobrança telefônica, e sistemas de pagamentos de salário.
5. *Sistemas de entretenimento*. São sistemas cuja utilização principal é pessoal e cujo objetivo é entreter o usuário. A maioria desses sistemas é de jogos de diferentes tipos. A qualidade de interação com o usuário é a característica particular mais importante dos sistemas de entretenimento.
6. *Sistemas para modelagem e simulação*. São sistemas que incluem vários objetos separados que interagem entre si, desenvolvidos por cientistas e engenheiros para modelar processos ou situações físicas. Esses sistemas geralmente fazem uso intensivo de recursos computacionais e requerem sistemas paralelos de alto desempenho para executar.
7. *Sistemas de coleta de dados*. São sistemas que coletam dados de seu ambiente com um conjunto de sensores e enviam esses dados para outros sistemas para processamento. O software precisa interagir com sensores e frequentemente é instalado em um ambiente hostil, por exemplo, dentro de uma máquina ou em um lugar remoto.
8. *Sistemas de sistemas*. São sistemas compostos de uma série de outros sistemas de software. Alguns deles podem ser produtos genéricos de software, como um programa de planilha eletrônica. Outros sistemas do conjunto podem ser escritos especialmente para esse ambiente.

É claro que as fronteiras entre esses tipos de sistema não são claras. Se você desenvolve um jogo para um telefone celular, deve levar em conta as mesmas restrições (energia, interação com hardware) que os desenvolvedores do software do telefone. Sistemas de processamento de lotes são frequentemente usados em conjunto com sistemas Web. Por exemplo, as requisições para reembolso das viagens dentro de uma empresa podem ser submetidas por meio de uma aplicação Web, porém processadas por uma aplicação de processamento de lotes para pagamento mensal.

Utilizamos diferentes técnicas de engenharia de software para cada tipo de sistema, porque cada software tem características bastante diversas. Por exemplo, um sistema de controle embutido em um automóvel é de segurança crítica e é gravado em memória ROM quando instalado no veículo. Por isso, sua alteração é muito cara. Tal sistema necessita de verificação e validação muito extensas para que as chances de ter de fazer um *recall* de carros depois de vendidos para correção de software sejam minimizadas. A interação do usuário, por sua vez, é mínima (ou talvez até inexistente), então não há necessidade de um processo de desenvolvimento que se baseie em prototipação de telas.

Para um sistema Web, uma abordagem baseada em desenvolvimento e entregas iterativas pode ser adequada, com o sistema sendo composto a partir de componentes reusáveis. No entanto, tal abordagem pode ser inviável para um sistema de sistemas, no qual as especificações detalhadas das interações do sistema precisam estar detalhadas antes para que cada sistema possa ser desenvolvido separadamente.

Apesar disso, existem fundamentos de engenharia de software que se aplicam a todos os tipos de sistemas de software:

1. Eles devem ser desenvolvidos em um processo gerenciado e compreendido. A organização que desenvolve o software deve planejar o processo de desenvolvimento e ter ideias claras do que será produzido e quando estará finalizado. É claro que processos diferentes são usados para tipos de software diferentes.
2. Confiança e desempenho são importantes para todos os tipos de sistema. O software deve se comportar conforme o esperado, sem falhas, e deve estar disponível para uso quando requerido. Deve ser seguro em sua operação e deve ser, tanto quanto possível, protegido contra ataques externos. O sistema deve executar de forma eficiente e não deve desperdiçar recursos.
3. É importante entender e gerenciar a especificação e os requisitos de software (o que o software deve fazer). Você deve saber o que clientes e usuários esperam dele e deve gerenciar suas expectativas para que um sistema útil possa ser entregue dentro do orçamento e do cronograma.
4. Você deve fazer o melhor uso possível dos recursos existentes. Isso significa que, quando apropriado, você deve reusar o software já desenvolvido, em vez de escrever um novo.

Essas noções básicas de processo, confiança, requisitos, gerenciamento e reúso são temas importantes desta obra. São refletidas de várias maneiras por diferentes métodos, e são o fundamento de todo o desenvolvimento de software profissional.

Você deve observar que esses fundamentos não cobrem implementação e programação. Eu não cubro técnicas de programação específicas neste livro, porque elas variam dramaticamente de um tipo de sistema para outro. Por exemplo, uma linguagem de *script* como Ruby é usada para programação de sistemas Web, porém seria totalmente inadequada para engenharia de sistemas embutidos.

1.1.3 Engenharia de software e a Internet

O desenvolvimento da Internet teve efeito profundo em nossas vidas. No início, a Internet era basicamente um armazenamento de informações acessível universalmente e tinha pouco efeito nos sistemas de software. Esses sistemas executavam em computadores locais e eram acessíveis apenas dentro da organização. Por volta do ano 2000, a Internet começou a evoluir, e mais e mais recursos passaram a ser adicionados aos navegadores. Isso significa que sistemas Web poderiam ser desenvolvidos e que, em vez de ter uma interface de usuário específica, poderiam ser acessados por um navegador. Isso levou ao desenvolvimento de uma enorme quantidade de novos produtos de software que ofereciam serviços inovadores e que eram acessados através da Internet. Esses produtos eram frequentemente sustentados pela propaganda exibida na tela do usuário e não exigiam pagamento direto.

Assim como esses produtos de software, o desenvolvimento de navegadores Web capazes de executar programas pequenos e fazer algum processamento local levou a uma evolução no software corporativo e organizacional. Em vez de escrever o software e instalá-lo nos computadores dos usuários, o software era implantado em um servidor Web. Isso tornou muito mais barato alterar e atualizar o software, porque não havia necessidade de se instalar o software em cada computador. Isso também reduziu os custos, porque o desenvolvimento de interface de usuário é particularmente caro. Consequentemente, sempre que possível, muitos negócios mudaram para interação Web com os sistemas de software da empresa.

O próximo estágio no desenvolvimento de sistemas Web foi a noção de *web services*. *Web services* são componentes de software acessados pela Internet e fornecem uma funcionalidade específica e útil. Aplicações são construídas integrando esses *web services*, os quais podem ser fornecidos por empresas diferentes. A princípio, essa ligação pode ser dinâmica, para que a aplicação possa usar *web services* diferentes toda vez que é executada. Essa abordagem para desenvolvimento de software é discutida no Capítulo 19.

Nos últimos anos, desenvolveu-se a ideia de 'software como serviço'. Foi proposto que o software normalmente não executará em computadores locais, e sim em 'nuvens computacionais' acessadas pela Internet. Se você usa um serviço como um webmail, está usando um sistema baseado em nuvem. Uma nuvem computacional consiste em um grande número de sistemas computacionais interligados, os quais são compartilhados entre vários usuários.

Os usuários não compram o software, mas pagam de acordo com o uso ou possuem acesso gratuito em troca de propagandas que são exibidas em suas telas.

Portanto, o surgimento da Internet trouxe uma mudança significativa na maneira como o software corporativo é organizado. Antes da Internet, aplicações corporativas eram, na maioria das vezes, monolíticas, programas isolados executando em computadores isolados ou em *clusters* de computadores. Agora, um software é altamente distribuído, às vezes pelo mundo todo. As aplicações corporativas não são programadas do zero; de fato, elas envolvem reúso extensivo de componentes e programas.

Essa mudança radical na organização de software obviamente causou mudanças na maneira como os sistemas Web são projetados. Por exemplo:

1. O reúso de software tornou-se a abordagem dominante para a construção de sistemas Web. Quando construímos esses sistemas, pensamos em como podemos montá-los a partir de componentes e sistemas de software preexistentes.
2. Atualmente, aceita-se que é impraticável especificar todos os requisitos para tais sistemas antecipadamente. Sistemas Web devem ser desenvolvidos e entregues incrementalmente.
3. Interfaces de usuário são restringidas pela capacidade dos navegadores. Embora tecnologias como AJAX (HOLDENER, 2008) signifiquem que interfaces ricas podem ser criadas dentro de um navegador, essas tecnologias ainda são difíceis de usar. Formulários Web com *scripts* locais são mais usados. Interfaces das aplicações em sistemas Web são normalmente mais pobres do que interfaces projetadas especialmente para produtos de software que executam em PCs.

As ideias fundamentais da engenharia de software discutidas na seção anterior aplicam-se para software baseado em Web da mesma forma que para outros tipos de sistemas de software. A experiência adquirida com o desenvolvimento de grandes sistemas no século XX ainda é relevante para softwares baseados em Web.

1.2 Ética na engenharia de software

Assim como outras disciplinas de engenharia, a engenharia de software é desenvolvida dentro de um *framework* social e legal que limita a liberdade das pessoas que trabalham nessa área. Como um engenheiro de software, você deve aceitar que seu trabalho envolve maiores responsabilidades do que simplesmente aplicar habilidades técnicas. Você também deve se comportar de forma ética e moralmente responsável se deseja ser respeitado como um engenheiro profissional.

Isso sem falar que você deve manter padrões normais de honestidade e integridade. Você não deve usar suas habilidades e seu conhecimento para se comportar de forma desonesta ou de maneira que possa denegrir a profissão de engenharia de software. No entanto, existem áreas nas quais os padrões de comportamento aceitável não são limitados pelas leis, mas pela mais tênue noção de responsabilidade profissional. Algumas delas são:

1. *Confidencialidade*. Você deve respeitar naturalmente a confidencialidade de seus empregadores ou clientes, independentemente de ter sido ou não assinado um acordo formal de confidencialidade.
2. *Competência*. Você não deve deturpar seu nível de competência. Você não deve aceitar conscientemente um trabalho que esteja fora de sua competência.
3. *Direitos de propriedade intelectual*. Você deve ter conhecimento das leis locais a respeito da propriedade intelectual, como patentes e copyright. Você deve ter cuidado para garantir que a propriedade intelectual dos empregadores e clientes seja protegida.
4. *Mau uso do computador*. Você não deve usar suas habilidades técnicas para fazer mau uso de computadores de outras pessoas. Esse mau uso varia de relativamente trivial (jogar videogames em uma máquina do empregador, por exemplo) até extremamente sério (disseminar vírus ou outros *malwares*).

Sociedades e instituições profissionais têm um papel importante a desempenhar na definição de padrões éticos. Organizações como ACM, IEEE (Institute of Eletrical and Electronic Engineers) e British Computer Society publicam um código de conduta profissional ou código de ética. Membros dessas organizações se comprometem a seguir esse código quando se tornam membros. Esses códigos de conduta normalmente se preocupam com o comportamento ético básico.

Associações profissionais, principalmente ACM e IEEE, cooperaram para produzir o código de ética e práticas profissionais. Esse código existe tanto na forma reduzida, mostrada no Quadro 1.1, quanto na forma completa

(GOTTERBARN et al., 1999), a qual acrescenta detalhes e conteúdo à versão resumida. O raciocínio por trás desse código está sumarizado nos dois primeiros parágrafos da versão completa:

> *Computadores têm um papel central e crescente no comércio, na indústria, no governo, na medicina, na educação, no entretenimento e na sociedade de um modo geral. Os engenheiros de software são aqueles que contribuem com a participação direta, ou lecionando, para análise, especificação, projeto, desenvolvimento, certificação, manutenção e testes de sistemas de software. Por causa de seu papel no desenvolvimento de sistemas de software, os engenheiros de software têm diversas oportunidades para fazer o bem ou para causar o mal, possibilitar que outros façam o bem ou causem o mal ou influenciar os outros a fazer o bem ou causar o mal. Para garantir ao máximo que seus esforços serão usados para o bem, os engenheiros de software devem se comprometer a fazer da engenharia de software uma profissão benéfica e respeitada. De acordo com esse compromisso, os engenheiros de software devem aderir ao Código de Ética e Prática Profissional a seguir.*
>
> *O Código contém oito princípios relacionados ao comportamento e às decisões dos engenheiros de software profissionais, incluindo praticantes, educadores, gerentes, supervisores e criadores de políticas, assim como trainees e estudantes da profissão. Esses princípios identificam os relacionamentos eticamente responsáveis dos quais cada indivíduo, grupo e organização participa e as principais obrigações dentro desses relacionamentos. As cláusulas de cada princípio são ilustrações de algumas obrigações inclusas nesses relacionamentos. Essas obrigações se baseiam na humanidade do engenheiro de software, especialmente no cuidado devido às pessoas afetadas pelo trabalho dos engenheiros de software e em elementos próprios da prática de engenharia de software. O Código prescreve essas obrigações como de qualquer um que se diz ser ou pretende ser um engenheiro de software.*

Em qualquer situação em que pessoas diferentes têm visões e objetivos diferentes, é provável que se enfrentem dilemas éticos. Por exemplo, se você discordar, em princípio, das políticas do gerenciamento de nível mais alto da empresa, como deve agir? É óbvio que isso depende das pessoas envolvidas e da natureza do desacordo. É melhor sustentar sua posição dentro da organização ou demitir-se por princípio? Se você acha que há problemas com um projeto de software, quando deve revelar isso à gerência? Se você discutir isso enquanto existem apenas suspeitas, poderá estar exagerando; se deixar para muito depois, poderá ser impossível resolver as dificuldades.

Tais dilemas éticos acontecem com todos nós em nossas vidas profissionais e, felizmente, na maioria dos casos eles são relativamente pequenos ou podem ser resolvidos sem muitas dificuldades. Quando não podem ser resolvidos, o engenheiro enfrenta, talvez, outro problema. A ação baseada em princípios pode ser pedir demissão, mas isso pode afetar outras pessoas, como seus(suas) companheiros(as) e filhos.

Quadro 1.1 Código de ética da ACM/IEEE (© IEEE/ACM 1999)

Código de ética e práticas profissionais da engenharia de software

Força-tarefa conjunta da ACM/IEEE-CS para ética e práticas profissionais da engenharia de software

Prefácio

Esta versão reduzida do código resume as aspirações em um alto nível de abstração; as cláusulas que estão inclusas na versão completa fornecem exemplos e detalhes de como essas aspirações mudam a forma como agimos enquanto profissionais de engenharia de software. Sem as aspirações, os detalhes podem se tornar legalistas e tediosos; sem os detalhes, as aspirações podem se tornar altissonantes, porém vazias; juntos, as aspirações e os detalhes formam um código coeso.

Os engenheiros de software devem se comprometer a fazer da análise, especificação, projeto, desenvolvimento, teste e manutenção de software uma profissão benéfica e respeitada. Em conformidade com seu comprometimento com a saúde, a segurança e o bem-estar públicos, engenheiros de software devem aderir a oito princípios:

1. PÚBLICO — Engenheiros de software devem agir de acordo com o interesse público.
2. CLIENTE E EMPREGADOR — Engenheiros de software devem agir de maneira que seja do melhor interesse de seu cliente e empregador e de acordo com o interesse público.
3. PRODUTO — Engenheiros de software devem garantir que seus produtos e modificações relacionadas atendam aos mais altos padrões profissionais possíveis.
4. JULGAMENTO — Engenheiros de software devem manter a integridade e a independência em seu julgamento profissional.
5. GERENCIAMENTO — Gerentes e líderes de engenharia de software devem aceitar e promover uma abordagem ética para o gerenciamento de desenvolvimento e manutenção de software.
6. PROFISSÃO — Engenheiros de software devem aprimorar a integridade e a reputação da profissão de acordo com o interesse público.
7. COLEGAS — Engenheiros de software devem auxiliar e ser justos com seus colegas.
8. SI PRÓPRIO — Engenheiros de software devem participar da aprendizagem contínua durante toda a vida, e devem promover uma abordagem ética para a prática da profissão.

Uma situação particularmente difícil para engenheiros profissionais aparece quando seu empregador age de forma antiética. Digamos que a empresa seja responsável por desenvolver um sistema de missão crítica e, por causa da pressão pelos prazos, falsifique os registros de validação de segurança. A responsabilidade do engenheiro é manter a confidencialidade, alertar o cliente ou divulgar, de alguma forma, que o sistema pode não ser seguro?

O problema aqui é que não há valores abslutos quando se trata de segurança. Embora o sistema possa não ter sido validado de acordo com os critérios predefinidos, esses critérios podem ser rígidos demais. O sistema pode, de fato, operar com segurança durante todo seu ciclo de vida. Também ocorre que, mesmo adequadamente validado, o sistema pode falhar e causar um acidente. A divulgação antecipada dos problemas pode resultar em prejuízo para o empregador e outros empregados; não divulgar os problemas pode resultar em prejuízo para outros.

Você deve tomar as próprias decisões em situações como essas. A postura ética adequada aqui depende totalmente dos pontos de vista dos indivíduos envolvidos. Nesse caso, o potencial do prejuízo, sua extensão e as pessoas afetadas por ele devem influenciar a decisão. Se a situação for muito perigosa, pode ser justificável divulgá-la usando a imprensa nacional (por exemplo). No entanto, você sempre deve tentar resolver a situação respeitando os direitos de seu empregador.

Outra questão ética é a participação no desenvolvimento de sistemas militares e nucleares. Algumas pessoas têm sentimentos fortes sobre essas questões e não desejam participar de qualquer desenvolvimento associado a sistemas militares. Outros trabalham em sistemas militares, mas não nos associados a armas. E ainda há aqueles que acham que a segurança nacional é um princípio fundamental e não têm objeções éticas em trabalhar em sistemas de armas.

Nessa situação, é importante que empregadores e empregados exponham sua visão uns aos outros antecipadamente. Quando uma organização está envolvida em um trabalho militar ou nuclear, ela deve ser capaz de deixar claro que os empregados devem estar dispostos a aceitar qualquer atribuição no trabalho. Igualmente, se qualquer empregado deixar claro que não deseja trabalhar em tais sistemas, os empregadores não devem pressioná-lo para fazer isso futuramente.

A área geral de ética e responsabilidade profissional está ficando mais importante à medida que os sistemas que fazem uso intensivo de software se infiltram em cada aspecto do trabalho e da vida cotidiana. Isso pode ser analisado do ponto de vista filosófico, em que os princípios básicos de ética são considerados, e a ética de engenharia de software é discutida com referência a esses princípios. Essa é a abordagem usada por Laudon (1995) e, em extensão menor, por Huff e Martin (1995). O artigo de Johnson sobre ética na computação (2001) também aborda o assunto de uma perspectiva filosófica.

No entanto, eu acho que essa abordagem filosófica é muito abstrata e difícil de ser relacionada com a experiência cotidiana. Eu prefiro uma abordagem mais concreta, baseada em códigos de conduta e práticas. Considero que a ética é mais bem discutida em um contexto de engenharia de software, e não como um assunto à parte. Portanto, não incluí neste livro discussões éticas abstratas, e sim, quando apropriado, exemplos nos exercícios que podem ser o ponto de partida para uma discussão em grupo sobre questões éticas.

1.3 Estudos de caso

Para ilustrar os conceitos de engenharia de software neste livro, uso exemplos de três diferentes tipos de sistemas. O motivo de não ter usado um único estudo de caso é que uma das mensagens-chave desta obra é que a prática da engenharia de software depende do tipo de sistema que está sendo produzido. Dessa forma, posso escolher um exemplo adequado quando discuto conceitos como segurança e confiança, modelagem de sistema, reúso etc.

Os três tipos de sistema que uso como estudos de caso são:

1. *Um sistema embutido.* Trata-se de um sistema no qual o software controla um dispositivo de hardware e é embutido nesse dispositivo. As questões em sistemas embutidos incluem tipicamente o tamanho físico, a capacidade de resposta, o gerenciamento de energia etc. O exemplo de um sistema embutido que uso é um sistema para controlar um dispositivo médico.
2. *Um sistema de informação.* Esse é um sistema cujo principal objetivo é gerenciar e prover acesso a um banco de dados de informações. As questões em sistemas de informação incluem proteção, usabilidade, privacidade

e manutenção da integridade dos dados. O exemplo de um sistema de informação que uso é um sistema de registros médicos.

3. *Um sistema de coleta de dados baseado em sensores*. Esse é um sistema cujo principal objetivo é coletar dados a partir de um conjunto de sensores e processá-los de alguma forma. Os principais requisitos de tais sistemas são confiabilidade, mesmo em condições ambientais hostis, e manutenibilidade. O exemplo de um sistema de coleta de dados que uso é uma estação meteorológica no deserto.

Apresento cada um desses sistemas neste capítulo, com mais informações a respeito de cada um deles disponíveis na Internet.

1.3.1 Sistema de controle de bomba de insulina

Uma bomba de insulina é um sistema médico que simula o funcionamento do pâncreas (um órgão interno). O software que controla o sistema é um sistema embutido, que coleta as informações a partir de um sensor e controla uma bomba que fornece uma dose controlada de insulina para o usuário.

Pessoas que sofrem de diabetes utilizam esse sistema. Diabetes é uma condição relativamente comum, na qual o pâncreas humano é incapaz de produzir quantidade suficiente de um hormônio chamado insulina. A insulina metaboliza glicose (açúcar) no sangue. O tratamento convencional de diabetes envolve injeções regulares de insulina sintetizada. Os diabéticos medem o nível de açúcar no sangue com um medidor externo, e depois calculam a dose de insulina que devem injetar.

O problema com esse tratamento é que o nível requerido de insulina não depende apenas do nível de glicose no sangue, mas também do tempo desde a última injeção. Isso pode levar a níveis muito baixos de glicose no sangue (se houver insulina demais) ou níveis muito altos de açúcar no sangue (se houver muito pouca insulina). Glicose baixa no sangue é, resumidamente, uma condição mais séria, porque pode resultar em mau funcionamento temporário do cérebro e, em casos extremos, inconsciência e morte. A longo prazo, no entanto, níveis altos contínuos de glicose no sangue podem causar prejuízos aos olhos, aos rins e problemas de coração.

Os avanços atuais no desenvolvimento de sensores miniaturizados possibilitaram a criação de sistemas automatizados de fornecimento de insulina. Esses sistemas monitoram o nível de açúcar no sangue e fornecem uma dose adequada de insulina quando necessário. Sistemas de fornecimento de insulina como esse já existem para o tratamento de pacientes hospitalares. No futuro, será possível para muitos diabéticos ter tais sistemas instalados permanentemente no corpo.

Um sistema de fornecimento de insulina controlado por software pode funcionar com o uso de um microssensor embutido no paciente para medir algum parâmetro do sangue que seja proporcional ao nível de açúcar. A informação coletada é então enviada para o controlador da bomba. Esse controlador calcula o nível de sangue e a quantidade necessária de insulina. Depois, um sinal é enviado para a bomba miniaturizada para fornecer a insulina através de uma agulha permanente.

A Figura 1.1 mostra os componentes de hardware e a organização da bomba de insulina. Para compreender os exemplos deste livro, tudo o que você precisa saber é que o sensor de sangue mede a condutividade elétrica do sangue em diferentes condições, e que esses valores podem ser relacionados ao nível de açúcar no sangue. A bomba de insulina fornece uma unidade de insulina como resposta a um único pulso do controlador. Portanto, para fornecer dez unidades de insulina, o controlador envia dez pulsos à bomba. A Figura 1.2 é um modelo de atividade UML (linguagem de modelagem unificada, do inglês *unified modeling language*), que ilustra como o software transforma uma entrada de nível de açúcar no sangue em uma sequência de comandos que operam a bomba de insulina.

É óbvio que esse é um sistema crítico de segurança. Se a bomba falhar a saúde do usuário pode ser prejudicada ou ele pode entrar em coma, porque o nível de açúcar em seu sangue estará muito alto ou muito baixo. Existem, portanto, dois requisitos essenciais a que esse sistema deve atender:

1. O sistema deve estar disponível para fornecer a insulina quando requerido.
2. O sistema deve executar de forma confiável e fornecer a quantidade correta de insulina para controlar o nível de açúcar no sangue.

Portanto, o sistema deve ser projetado e implementado para garantir que atenda sempre a esses requisitos. Requisitos e discussões mais detalhados sobre como garantir a segurança do sistema são discutido mais adiante.

Figura 1.1 Hardware de bomba de insulina

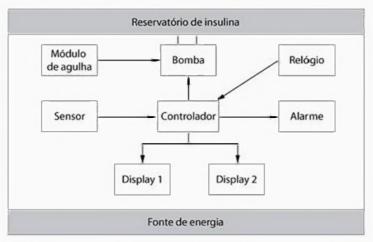



Figura 1.2 Modelo de atividade da bomba de insulina

Sensor de sangue
Analisar a leitura do sensor
Açúcar no sangue
Calcular insulina
Registro de insulina
Dose de insulina
Processar comandos da bomba
Registro de dose
Dados da bomba
Controlar bomba de insulina
Bomba de insulina

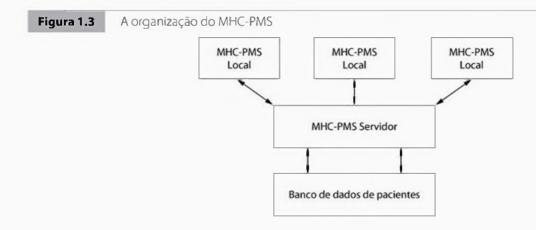
1.3.2 Um sistema de informação de pacientes para cuidados com saúde mental

Um sistema de informação de pacientes para apoiar cuidados com saúde mental é um sistema médico de informação que mantém os dados sobre os pacientes que sofrem dos problemas de saúde mental e os tratamentos que eles receberam. A maioria dos pacientes com problemas mentais não requer tratamento hospitalar dedicado; em geral, eles precisam visitar clínicas especializadas regularmente, onde podem encontrar um médico que possua conhecimento detalhado de seus problemas. Para ficar mais fácil atender os pacientes, essas clínicas não existem apenas dentro dos hospitais. Elas também podem ser encontradas em centros comunitários de saúde.

O MHC-PMS (sistema de gerenciamento de pacientes com problemas de saúde mental, do inglês *menthal health care-patient management system*) é um sistema de informação utilizado em tais clínicas. Ele usa um banco de dados centralizado de informações dos pacientes. Mas esse sistema também foi projetado para executar em um PC, para poder ser usado em ambientes que não possuem uma conexão de rede segura. Quando o sistema local possui um acesso de rede seguro, a informação sobre os pacientes é usada a partir do banco de dados, mas essa informação pode ser baixada, e uma cópia local de registros de pacientes pode ser usada quando não há conexão disponível. O sistema não é um sistema completo de registros médicos e, por isso, não contém informações sobre outras condições médicas. A Figura 1.3 ilustra a organização do MHC-PMS.

O MHC-PMS tem dois objetivos principais:

1. Gerar informação gerencial que permita aos gestores do serviço de saúde avaliar o desempenho de alvos locais e governamentais.
2. Fornecer ao pessoal médico informação atualizada para apoiar o tratamento dos pacientes.

Figura 1.3 A organização do MHC-PMS

A natureza de problemas de saúde mental tem como característica o fato de os pacientes serem, frequentemente, desorganizados e perderem seus compromissos, proposital ou acidentalmente, bem como receitas e remédios, esquecerem as instruções e demandarem atendimento médico de forma exagerada. Esses pacientes podem aparecer nas clínicas inesperadamente. Na minoria dos casos, podem se tornar um perigo para si mesmos ou para outras pessoas. Podem mudar de endereço regularmente ou estar desabrigados há muito ou pouco tempo. Quando os pacientes são perigosos, podem precisar de internação – em um hospital seguro, para tratamento e observação.

Usuários do sistema incluem o pessoal da clínica, como médicos, enfermeiros e agentes de saúde (enfermeiros que visitam as pessoas em casa para verificar seu tratamento). Usuários que não são da área médica incluem os recepcionistas que agendam as consultas, o pessoal que faz manutenção nos cadastros do sistema e o pessoal administrativo que gera os relatórios.

O sistema é usado para guardar a informação sobre os pacientes (nome, endereço, idade, parente mais próximo etc.), consultas (data, médico que atendeu, impressões subjetivas sobre o paciente etc.), condições e tratamentos. Relatórios são gerados em intervalos regulares para o pessoal médico e gestores de saúde autorizados. Normalmente, os relatórios para o pessoal médico focam a informação sobre pacientes individuais, enquanto relatórios gerenciais são anônimos e se preocupam com condições, custos dos tratamentos etc.

Os principais recursos do sistema são:

1. *Gerenciamento do cuidado individual.* O pessoal clínico pode criar registros dos pacientes, editar as informações no sistema, ver histórico dos pacientes etc. O sistema suporta resumo de dados, para que os médicos que não conheceram o paciente anteriormente possam conhecer rapidamente seus principais problemas e os tratamentos prescritos.
2. *Monitoramento de pacientes.* O sistema regularmente monitora os registros dos pacientes que são envolvidos nos tratamentos e emite alertas quando possíveis problemas são detectados. Portanto, se o paciente não se encontrou com o médico durante algum tempo, um alerta pode ser emitido. Um dos elementos mais importantes do sistema de monitoramento é manter registro dos pacientes que foram internados e garantir que as verificações legais sejam efetuadas em tempo certo.
3. *Relatórios administrativos.* O sistema gera relatórios gerencias mensais mostrando o número de pacientes tratados em cada clínica, o número de pacientes que entraram e saíram do sistema de saúde, o número de pacientes internados, os remédios prescritos e seus custos etc.

Duas leis diferentes afetam o sistema. A lei de proteção de dados que governa a confidencialidade da informação pessoal e a lei de saúde mental, que governa a detenção compulsória de pacientes considerados perigosos para si mesmos ou para outros. A saúde mental diferencia-se das demais especialidades por ser a única que permite a um médico recomendar a internação de um paciente contra sua vontade. Isso está sujeito a garantias legislativas muito rigorosas. Um dos objetivos do MHC-PMS é garantir que o pessoal sempre aja de acordo com as leis e que suas decisões sejam registradas para fiscalização judicial, caso necessário.

Assim como acontece em todos os sistemas médicos, a privacidade é um requisito crítico do sistema. É essencial que a informação do paciente seja confidencial e que jamais seja revelada a ninguém além do pessoal médico autorizado e dos próprios pacientes. O MHC-PMS é também um sistema de segurança crítica. Algumas doenças

mentais fazem com que os pacientes se tornem suicidas ou perigosos para outras pessoas. Sempre que possível, o sistema deve alertar o pessoal médico sobre pacientes potencialmente suicidas ou perigosos.

O projeto geral do sistema deve levar em conta os requisitos de privacidade e segurança. O sistema deve estar disponível quando necessário; caso contrário, a segurança pode ficar comprometida e pode ficar impossível prescrever a medicação correta para os pacientes. Existe um conflito em potencial aqui — é mais fácil manter a privacidade quando existe apenas uma cópia de dados do sistema; no entanto, para garantir a disponibilidade no caso de falha de servidor ou quando não houver conexão de rede, múltiplas cópias de dados devem ser mantidas. Discuto os compromissos entre esses requisitos nos próximos capítulos.

1.3.3 Uma estação meteorológica no deserto

Para ajudar a monitorar as mudanças climáticas e aprimorar a exatidão de previsões do tempo em áreas distantes, o governo de um país com grandes áreas desertas decide instalar centenas de estações meteorológicas em áreas remotas. Essas estações meteorológicas coletam dados a partir de um conjunto de instrumentos que medem temperatura, pressão, sol, chuva, velocidade e direção do vento.

As estações meteorológicas no deserto são parte de um sistema maior (Figura 1.4), um sistema de informações meteorológicas que coleta dados a partir das estações meteorológicas e os disponibiliza para serem processados por outros sistemas. Os sistemas da Figura 1.4 são:

1. *Sistema da estação meteorológica*. Responsável por coletar dados meteorológicos, efetuar algum processamento inicial de dados e transmiti-los para o sistema de gerenciamento de dados.
2. *Sistema de gerenciamento e arquivamento de dados*. Esse sistema coleta os dados de todas as estações meteorológicas, executa o processamento e a análise dos dados e os arquiva de forma que possam ser obtidos por outros sistemas, como sistemas de previsões de tempo.
3. *Sistema de manutenção da estação*. Esse sistema pode se comunicar via satélite com todas as estações meteorológicas no deserto para monitorar as condições desses sistemas e fornecer relatórios sobre os problemas. Ele também pode atualizar o software embutido nesses sistemas. Em caso de problema com o sistema ele pode, ainda, ser usado para controlar remotamente um sistema meteorológico no deserto.

Na Figura 1.4 utilizei o símbolo de pacote da UML para indicar que cada sistema é uma coleção de componentes e identifiquei sistemas separados, usando o estereótipo <<sistema>> da UML. As associações entre os pacotes indicam que há uma troca de informações, mas, nesse estágio, não há necessidade de defini-la com mais detalhes.

Cada estação meteorológica inclui uma série de instrumentos que medem os parâmetros do tempo, como velocidade e direção do vento, temperatura do solo e do ar, pressão barométrica e chuva em um período de 24 horas. Cada um desses instrumentos é controlado por um sistema de software que obtém periodicamente a leitura dos parâmetros e gerencia os dados coletados a partir desses instrumentos.

O sistema da estação meteorológica opera coletando as observações do tempo em intervalos frequentes — por exemplo, temperaturas são medidas a cada minuto. No entanto, como a largura de banda da conexão por satélite é relativamente insuficiente, a estação meteorológica efetua algum processamento local de agregação de dados. A transmissão desses dados agregados ocorre a partir da solicitação do sistema coletor. Se, por um motivo qualquer, for impossível estabelecer a conexão, a estação meteorológica mantém os dados localmente até se restabelecer a comunicação.

Figura 1.4 Ambiente de estação meteorológica

«sistema»
Estação meteorológica

«sistema»
Gerenciamento
e arquivamento de dados

«sistema»
Manutenção da estação

Cada estação meteorológica tem uma bateria que a alimenta e deve ser totalmente autocontida — não há energia externa ou cabos de rede disponíveis. Todas as comunicações passam por um link via satélite relativamente lento, e a estação meteorológica deve incluir algum mecanismo (solar ou eólico) para recarregar as baterias. Por serem instaladas em áreas desertas, elas são expostas a diversas condições ambientais e podem ser avariadas por animais. O software da estação não se preocupa apenas com coleção de dados. Ele deve também:

1. Monitorar os instrumentos, energia e hardware de comunicação e reportar defeitos para o sistema de gerenciamento.
2. Gerenciar a energia do sistema, garantindo o carregamento das baterias sempre que as condições ambientais permitirem, bem como o desligamento dos geradores em condições climáticas potencialmente perigosas, como ventos fortes.
3. Permitir reconfiguração dinâmica quando partes do software forem substituídas com novas versões e quando os instrumentos de backup forem conectados ao sistema em caso de falha de sistema.

Como as estações meteorológicas precisam ser autocontidas e independentes, o software instalado é complexo, apesar de a funcionalidade de coleta de dados ser bastante simples.

- Engenharia de software é uma disciplina de engenharia que se preocupa com todos os aspectos da produção de software.
- Software não é apenas um programa ou programas; ele inclui também a documentação. Os atributos principais de um produto de software são manutenibilidade, confiança, proteção, eficiência e aceitabilidade.
- O processo de software inclui todas as atividades envolvidas no desenvolvimento do software. Atividades de alto nível de especificação, desenvolvimento, validação e evolução são parte de todos os processos de software.
- As ideias fundamentais da engenharia de software são universalmente aplicáveis para todos os tipos de desenvolvimento de sistemas. Esses fundamentos incluem processos de software, confiança, proteção, requisitos e reúso.
- Existem vários tipos diferentes de sistemas, e cada um requer ferramentas e técnicas de engenharia de software adequadas a seu desenvolvimento. Existem poucas, se houver alguma, técnicas específicas de projeto e implementação aplicáveis para todos os tipos de sistemas.
- As ideias básicas da engenharia de software são aplicáveis a todos os tipos de sistemas de software. Esses fundamentos incluem processos de software gerenciados, confiança e proteção de software, engenharia de requisitos e reúso de software.
- Engenheiros de software têm responsabilidades com a profissão de engenharia e com a sociedade. Eles não devem se preocupar apenas com as questões técnicas.
- Sociedades profissionais publicam códigos de conduta que definem os padrões de comportamento esperado de seus membros.

"No silver bullet: Essence and accidents of software engineering". Apesar de não ser recente, esse artigo é uma boa introdução geral para os problemas da engenharia de software. A mensagem essencial do artigo ainda não mudou. (BROOKS, F. P. *IEEE Computer*, v. 20, n. 4, abr. 1987.) Disponível em: <http://doi.ieeecomputersociety.org/10.1109/MC.1987.1663532>.

"Software engineering code of ethics is approved". Esse artigo discute os fundamentos do desenvolvimento do Código de Ética de ACM/IEEE, e inclui ambas as formas do código, resumida e longa. (GOTTERBARN, D.; MILLER, K.; ROGERSON S. *Comm. ACM*, vol. 42, n. 10, out. 1999.) Disponível em: <http://portal.acm.org/citation.cfm?doid=317665.317682>.

Professional Issues in Software Engineering. Esse é um excelente livro que discute questões legais e profissionais, assim como as éticas. Eu prefiro sua abordagem prática a textos mais teóricos sobre ética. (BOTT, F.; COLEMAN, A.; EATON, J.; ROWLAND, D. *Professional Issues in Software Engineering*. 3. ed. Londres e Nova York: Taylor and Francis, 2000.)

IEEE Software, Março/Abril de 2002. Essa é uma edição especial da revista dedicada ao desenvolvimento de software para Internet. Essa área mudou muito rapidamente, então alguns artigos estão um pouco ultrapassados, mas a maioria ainda é relevante. (*IEEE Software*, v. 19, n. 2, mar./abr. 2002.) Disponível em: <http://www2.computer.org/portal/web/software>.

"A View of 20th and 21st Century Software Engineering". Uma visão do passado e do futuro da engenharia de software escrita por um dos primeiros e mais respeitados engenheiros de software. Barry Boehm identifica princípios da engenharia de software que não mudam com o passar do tempo, mas também sugere que algumas das práticas comuns sejam obsoletas. (BOEHM, B. *28th Software Engineering Conf.*, Shanghai. 2006.) Disponível em: <http://doi.ieeecomputersociety.org/10.1145/1134285.1134288>.

"Software Engineering Ethics". Edição especial de *IEEE Computer*, com uma série de artigos sobre o assunto. (*IEEE Computer*, v. 42, n. 6, jun. 2009.)

EXERCÍCIOS

1.1 Explique por que software profissional não é apenas os programas que são desenvolvidos para o cliente.

1.2 Qual a diferença mais importante entre o desenvolvimento de um produto genérico de software e o desenvolvimento de software sob demanda? O que isso pode significar na prática para usuários de produtos de software genérico?

1.3 Quais são os quatro atributos importantes que todo software profissional deve possuir? Sugira outros quatro atributos que, às vezes, podem ser significantes.

1.4 Além dos desafios de heterogeneidade, mudanças sociais e corporativas, confiança e proteção, identifique outros problemas e desafios que a engenharia de software provavelmente enfrentará no século XXI (Dica: pense no meio ambiente).

1.5 Baseado em seu conhecimento de alguns tipos de aplicações discutidos na Seção 1.1.2, explique, com exemplos, por que tipos de aplicações diferentes requerem técnicas especializadas de engenharia de software para apoiar seu projeto e desenvolvimento.

1.6 Explique por que existem ideias fundamentais na engenharia de software que se aplicam a todos os tipos de sistemas.

1.7 Explique como o uso universal da Internet mudou os sistemas de software.

1.8 Discuta se os engenheiros profissionais devem ser certificados da mesma forma que médicos e advogados.

1.9 Para cada uma das cláusulas no Código de Ética da ACM/IEEE mostradas no Quadro 1.1, sugira um exemplo adequado para ilustrar.

1.10 Para ajudar a combater o terrorismo, muitos países estão planejando desenvolver, ou já desenvolveram, sistemas computacionais que rastreiam grandes números de cidadãos e suas ações. Obviamente, isso tem implicações nas questões da privacidade. Discuta a ética de se trabalhar desenvolvendo esse tipo de sistema.

REFERÊNCIAS

GOTTERBARN, D.; MILLER, K.; ROGERSON, S. Software Engineering Code of Ethics is Approved. *Comm. ACM*, v. 42, n. 10, 1999, p. 102-107.

HOLDENER, A. T. *Ajax: The Definitive Guide*. Sebastopol: O'Reilly and Associates, 2008.

HUFF, C.; MARTIN, C. D. Computing Consequences: A Framework for Teaching Ethical Computing. *Comm. ACM*, v. 38, n. 12, 1995, p. 75-84.

JOHNSON, D. G. *Computer Ethics*. Englewood Cliffs: Prentice Hall, 2001.

LAUDON, K. Ethical Concepts and Information Technology. *Comm. ACM*, v. 38, n. 12, 1995, p. 33-39.

NAUR, P.; RANDELL, B. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*. Garmisch, Germany. 7 a 11 out. 1968.

CAPÍTULO

1 **2** 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Processos de software

Objetivos

O objetivo deste capítulo é apresentar a ideia de um processo de software — um conjunto coerente de atividades para a produção de software. Ao terminar de ler este capítulo, você:

- compreenderá os conceitos e modelos de processos de software;
- terá sido apresentado a três modelos genéricos de processos de software e quando eles podem ser usados;
- conhecerá as atividades fundamentais do processo de engenharia de requisitos de software, desenvolvimento de software, testes e evolução;
- entenderá por que os processos devem ser organizados de maneira a lidar com as mudanças nos requisitos e projeto de software;
- compreenderá como o Rational Unified Process integra boas práticas de engenharia de software para criar processos de software adaptáveis.

Conteúdo

Um processo de software é um conjunto de atividades relacionadas que levam à produção de um produto de software. Essas atividades podem envolver o desenvolvimento de software a partir do zero em uma linguagem padrão de programação como Java ou C. No entanto, aplicações de negócios não são necessariamente desenvolvidas dessa forma. Atualmente, novos softwares de negócios são desenvolvidos por meio da extensão e modificação de sistemas existentes ou por meio da configuração e integração de prateleira ou componentes do sistema.

Existem muitos processos de software diferentes, mas todos devem incluir quatro atividades fundamentais para a engenharia de software:

1. *Especificação de software.* A funcionalidade do software e as restrições a seu funcionamento devem ser definidas.
2. *Projeto e implementação de software.* O software deve ser produzido para atender às especificações.
3. *Validação de software.* O software deve ser validado para garantir que atenda às demandas do cliente.
4. *Evolução de software.* O software deve evoluir para atender às necessidades de mudança dos clientes.

De alguma forma, essas atividades fazem parte de todos os processos de software. Na prática, são atividades complexas em si mesmas, que incluem subatividades como validação de requisitos, projeto de arquitetura, testes unitários etc. Existem também as atividades que dão apoio ao processo, como documentação e gerenciamento de configuração de software.

Ao descrever e discutir os processos, costumamos falar sobre suas atividades, como a especificação de um modelo de dados, o projeto de interface de usuário etc., bem como a organização dessas atividades. No entanto, assim como as atividades, as descrições do processo também podem incluir:

1. Produtos, que são os resultados de uma das atividades do processo. Por exemplo, o resultado da atividade de projeto de arquitetura pode ser um modelo da arquitetura de software.
2. Papéis, que refletem as responsabilidades das pessoas envolvidas no processo. Exemplos de papéis são: gerente de projeto, gerente de configuração, programador etc.
3. Pré e pós-condições, que são declarações verdadeiras antes e depois de uma atividade do processo ou da produção de um produto. Por exemplo, antes do projeto de arquitetura ser iniciado, pode haver uma pré-condição de que todos os requisitos tenham sido aprovados pelo cliente e, após a conclusão dessa atividade, uma pós-condição poderia ser a de que os modelos UML que descrevem a arquitetura tenham sido revisados.

Os processos de software são complexos e, como todos os processos intelectuais e criativos, dependem de pessoas para tomar decisões e fazer julgamentos. Não existe um processo ideal, a maioria das organizações desenvolve os próprios processos de desenvolvimento de software. Os processos têm evoluído de maneira a tirarem melhor proveito das capacidades das pessoas em uma organização, bem como das características específicas do sistema em desenvolvimento. Para alguns sistemas, como sistemas críticos, é necessário um processo de desenvolvimento muito bem estruturado; para sistemas de negócios, com requisitos que se alteram rapidamente, provavelmente será mais eficaz um processo menos formal e mais flexível.

Os processos de software, às vezes, são categorizados como dirigidos a planos ou processos ágeis. Processos dirigidos a planos são aqueles em que todas as atividades são planejadas com antecedência, e o progresso é avaliado por comparação com o planejamento inicial. Em processos ágeis, que discuto no Capítulo 3, o planejamento é gradativo, e é mais fácil alterar o processo de maneira a refletir as necessidades de mudança dos clientes. Conforme Boehm e Turner (2003), cada abordagem é apropriada para diferentes tipos de software. Geralmente, é necessário encontrar um equilíbrio entre os processos dirigidos a planos e os processos ágeis.

Embora não exista um processo 'ideal' de software, há espaço, em muitas organizações, para melhorias no processo de software. Os processos podem incluir técnicas ultrapassadas ou não aproveitar as melhores práticas de engenharia de software da indústria. De fato, muitas empresas ainda não se aproveitam dos métodos da engenharia de software em seu desenvolvimento de software.

Em organizações nas quais a diversidade de processos de software é reduzida, os processos de software podem ser melhorados pela padronização. Isso possibilita uma melhor comunicação, além de redução no período de treinamento, e torna mais econômico o apoio ao processo automatizado. A padronização também é um importante primeiro passo na introdução de novos métodos e técnicas de engenharia de software, assim como as boas práticas de engenharia de software. No Capítulo 26, discuto mais detalhadamente a melhoria no processo de software.

2.1 Modelos de processo de software

Como expliquei no Capítulo 1, um modelo de processo de software é uma representação simplificada de um processo de software. Cada modelo representa uma perspectiva particular de um processo e, portanto, fornece informações parciais sobre ele. Por exemplo, um modelo de atividade do processo pode mostrar as atividades e sua sequência, mas não mostrar os papéis das pessoas envolvidas. Nesta seção, apresento uma série de modelos gerais de processos (algumas vezes, chamados 'paradigmas de processo') a partir de uma perspectiva de sua arquitetura. Ou seja, nós vemos um *framework* do processo, mas não vemos os detalhes de suas atividades específicas.

Esses modelos genéricos não são descrições definitivas dos processos de software. Pelo contrário, são abstrações que podem ser usadas para explicar diferentes abordagens de desenvolvimento de software. Você pode vê-los como *frameworks* de processos que podem ser ampliados e adaptados para criar processos de engenharia de software mais específicos.

Os modelos de processo que abordo aqui são:

1. *O modelo em cascata.* Esse modelo considera as atividades fundamentais do processo de especificação, desenvolvimento, validação e evolução, e representa cada uma delas como fases distintas, como: especificação de requisitos, projeto de software, implementação, teste e assim por diante.

2. *Desenvolvimento incremental.* Essa abordagem intercala as atividades de especificação, desenvolvimento e validação. O sistema é desenvolvido como uma série de versões (incrementos), de maneira que cada versão adiciona funcionalidade à anterior.
3. *Engenharia de software orientada a reúso.* Essa abordagem é baseada na existência de um número significativo de componentes reusáveis. O processo de desenvolvimento do sistema concentra-se na integração desses componentes em um sistema já existente em vez de desenvolver um sistema a partir do zero.

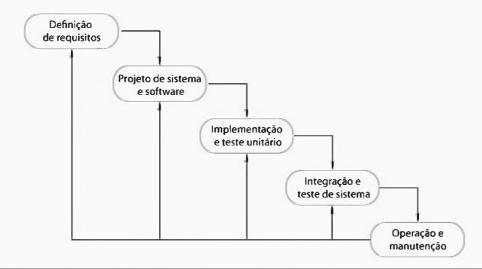
Esses modelos não são mutuamente exclusivos e muitas vezes são usados em conjunto, especialmente para o desenvolvimento de sistemas de grande porte. Para sistemas de grande porte, faz sentido combinar algumas das melhores características do modelo em cascata e dos modelos de desenvolvimento incremental. É preciso ter informações sobre os requisitos essenciais do sistema para projetar uma arquitetura de software que dê suporte a esses requisitos. Você não pode desenvolver isso incrementalmente. Os subsistemas dentro de um sistema maior podem ser desenvolvidos com diferentes abordagens. As partes do sistema que são bem compreendidas podem ser especificadas e desenvolvidas por meio de um processo baseado no modelo em cascata. As partes que são difíceis de especificar antecipadamente, como a interface com o usuário, devem sempre ser desenvolvidas por meio de uma abordagem incremental.

2.1.1 O modelo em cascata

O primeiro modelo do processo de desenvolvimento de software a ser publicado foi derivado de processos mais gerais da engenharia de sistemas (ROYCE, 1970). Esse modelo é ilustrado na Figura 2.1. Por causa do encadeamento entre uma fase e outra, esse modelo é conhecido como 'modelo em cascata', ou ciclo de vida de software. O modelo em cascata é um exemplo de um processo dirigido a planos — em princípio, você deve planejar e programar todas as atividades do processo antes de começar a trabalhar nelas.

Os principais estágios do modelo em cascata refletem diretamente as atividades fundamentais do desenvolvimento:

1. *Análise e definição de requisitos.* Os serviços, restrições e metas do sistema são estabelecidos por meio de consulta aos usuários. Em seguida, são definidos em detalhes e funcionam como uma especificação do sistema.
2. *Projeto de sistema e software.* O processo de projeto de sistemas aloca os requisitos tanto para sistemas de hardware como para sistemas de software, por meio da definição de uma arquitetura geral do sistema. O projeto de software envolve identificação e descrição das abstrações fundamentais do sistema de software e seus relacionamentos.
3. *Implementação e teste unitário.* Durante esse estágio, o projeto do software é desenvolvido como um conjunto de programas ou unidades de programa. O teste unitário envolve a verificação de que cada unidade atenda a sua especificação.

Figura 2.1 O modelo em cascata

4. *Integração e teste de sistema.* As unidades individuais do programa ou programas são integradas e testadas como um sistema completo para assegurar que os requisitos do software tenham sido atendidos. Após o teste, o sistema de software é entregue ao cliente.
5. *Operação e manutenção.* Normalmente (embora não necessariamente), essa é a fase mais longa do ciclo de vida. O sistema é instalado e colocado em uso. A manutenção envolve a correção de erros que não foram descobertos em estágios iniciais do ciclo de vida, com melhora da implementação das unidades do sistema e ampliação de seus serviços em resposta às descobertas de novos requisitos.

Em princípio, o resultado de cada estágio é a aprovação de um ou mais documentos ('assinados'). O estágio seguinte não deve ser iniciado até que a fase anterior seja concluída. Na prática, esses estágios se sobrepõem e alimentam uns aos outros de informações. Durante o projeto, os problemas com os requisitos são identificados; durante a codificação, problemas de projeto são encontrados e assim por diante. O processo de software não é um modelo linear simples, mas envolve o *feedback* de uma fase para outra. Assim, os documentos produzidos em cada fase podem ser modificados para refletirem as alterações feitas em cada um deles.

Por causa dos custos de produção e aprovação de documentos, as iterações podem ser dispendiosas e envolver significativo retrabalho. Assim, após um pequeno número de iterações, é normal se congelarem partes do desenvolvimento, como a especificação, e dar-se continuidade aos estágios posteriores de desenvolvimento. A solução dos problemas fica para mais tarde, ignorada ou programada, quando possível. Esse congelamento prematuro dos requisitos pode significar que o sistema não fará o que o usuário quer. Também pode levar a sistemas mal estruturados, quando os problemas de projeto são contornados por artifícios de implementação.

Durante o estágio final do ciclo de vida (operação e manutenção), o software é colocado em uso. Erros e omissões nos requisitos originais do software são descobertos. Os erros de programa e projeto aparecem e são identificadas novas necessidades funcionais. O sistema deve evoluir para permanecer útil. Fazer essas alterações (manutenção do software) pode implicar repetição de estágios anteriores do processo.

O modelo em cascata é consistente com outros modelos de processos de engenharia, e a documentação é produzida em cada fase do ciclo. Dessa forma, o processo torna-se visível, e os gerentes podem monitorar o progresso de acordo com o plano de desenvolvimento. Seu maior problema é a divisão inflexível do projeto em estágios distintos. Os compromissos devem ser assumidos em um estágio inicial do processo, o que dificulta que atendam às mudanças de requisitos dos clientes.

Em princípio, o modelo em cascata deve ser usado apenas quando os requisitos são bem compreendidos e pouco provavelmente venham a ser radicalmente alterados durante o desenvolvimento do sistema. No entanto, o modelo em cascata reflete o tipo de processo usado em outros projetos de engenharia. Como é mais fácil usar um modelo de gerenciamento comum para todo o projeto, processos de software baseados no modelo em cascata ainda são comumente utilizados.

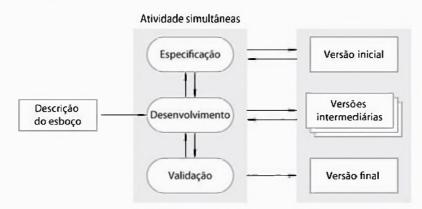
Uma variação importante do modelo em cascata é o desenvolvimento formal de um sistema, em que se cria um modelo matemático de uma especificação do sistema. Esse modelo é então refinado, usando transformações matemáticas que preservam sua consistência, em código executável. Partindo do pressuposto de que suas transformações matemáticas estão corretas, você pode, portanto, usar um forte argumento de que um programa gerado dessa forma é consistente com suas especificações.

Processos formais de desenvolvimento, como os baseados no método B (SCHNEIDER, 2001; WORDSWORTH, 1996), são particularmente adequados para o desenvolvimento de sistemas com requisitos rigorosos de segurança, confiabilidade e proteção. A abordagem formal simplifica a produção de casos de segurança ou proteção. Isso demonstra aos clientes ou reguladores que o sistema realmente cumpre com seus requisitos de proteção ou segurança.

Processos baseados em transformações formais são geralmente usados apenas no desenvolvimento de sistemas críticos de segurança ou de proteção. Eles exigem conhecimentos especializados. Para a maioria dos sistemas, esse processo não oferece custo-benefício significativo sobre outras abordagens para o desenvolvimento de sistemas.

2.1.2 Desenvolvimento Incremental

O desenvolvimento incremental é baseado na ideia de desenvolver uma implementação inicial, expô-la aos comentários dos usuários e continuar por meio da criação de várias versões até que um sistema adequado seja desenvolvido (Figura 2.2). Atividades de especificação, desenvolvimento e validação são intercaladas, e não separadas, com rápido *feedback* entre todas as atividades.

Figura 2.2 Desenvolvimento incremental

Desenvolvimento incremental de software, que é uma parte fundamental das abordagens ágeis, é melhor do que uma abordagem em cascata para a maioria dos sistemas de negócios, *e-commerce* e sistemas pessoais. Desenvolvimento incremental reflete a maneira como resolvemos os problemas. Raramente elaboramos uma completa solução do problema com antecedência; geralmente movemo-nos passo a passo em direção a uma solução, recuando quando percebemos que cometemos um erro. Ao desenvolver um software de forma incremental, é mais barato e mais fácil fazer mudanças no software durante seu desenvolvimento.

Cada incremento ou versão do sistema incorpora alguma funcionalidade necessária para o cliente. Frequentemente, os incrementos iniciais incluem a funcionalidade mais importante ou mais urgente. Isso significa que o cliente pode avaliar o sistema em um estágio relativamente inicial do desenvolvimento para ver se ele oferece o que foi requisitado. Em caso negativo, só o incremento que estiver em denvolvimento no momento precisará ser alterado e, possivelmente, nova funcionalidade deverá ser definida para incrementos posteriores.

O desenvolvimento incremental tem três vantagens importantes quando comparado ao modelo em cascata:

1. O custo de acomodar as mudanças nos requisitos do cliente é reduzido. A quantidade de análise e documentação a ser refeita é muito menor do que o necessário no modelo em cascata.
2. É mais fácil obter *feedback* dos clientes sobre o desenvolvimento que foi feito. Os clientes podem fazer comentários sobre as demonstrações do software e ver o quanto foi implementado. Os clientes têm dificuldade em avaliar a evolução por meio de documentos de projeto de software.
3. É possível obter entrega e implementação rápida de um software útil ao cliente, mesmo se toda a funcionalidade não for incluída. Os clientes podem usar e obter ganhos a partir do software inicial antes do que é possível com um processo em cascata.

O desenvolvimento incrementai, atualmente, é a abordagem mais comum para o desenvolvimento de sistemas aplicativos. Essa abordagem pode ser tanto dirigida a planos, ágil, ou, o mais comum, uma mescla dessas abordagens. Em uma abordagem dirigida a planos, os incrementos do sistema são identificados previamente; se uma abordagem ágil for adotada, os incrementos iniciais são identificados, mas o desenvolvimento de incrementos posteriores depende do progresso e das prioridades dos clientes.

Do ponto de vista do gerenciamento, a abordagem incremental tem dois problemas:

1. O processo não é visível. Os gerentes precisam de entregas regulares para mensurar o progresso. Se os sistemas são desenvolvidos com rapidez, não é economicamente viável produzir documentos que reflitam cada uma das versões do sistema.
2. A estrutura do sistema tende a se degradar com a adição dos novos incrementos. A menos que tempo e dinheiro sejam dispendidos em refatoração para melhoria do software, as constantes mudanças tendem a corromper sua estrutura. Incorporar futuras mudanças do software torna-se cada vez mais difícil e oneroso.

Os problemas do desenvolvimento incremental são particularmente críticos para os sistemas de vida-longa, grandes e complexos, nos quais várias equipes desenvolvem diferentes partes do sistema. Sistemas de grande porte necessitam de um *framework* ou arquitetura estável, e as responsabilidades das diferentes equipes de trabalho do sistema precisam ser claramente definidas, respeitando essa arquitetura. Isso deve ser planejado com antecedência, e não desenvolvido de forma incremental.

Você pode desenvolver um sistema de forma incrementai e expô-lo aos comentários dos clientes, sem realmente entregá-lo e implantá-lo no ambiente do cliente. Entrega e implantação incrementai significa que o software é usado em processos operacionais reais. Isso nem sempre é possível, pois experimentações com o novo software podem interromper os processos normais de negócios. As vantagens e desvantagens da entrega incrementai são discutidas na Seção 2.3.2.

2.1.3 Engenharia de software orientada a reúso

Na maioria dos projetos de software, há algum reúso de software. Isso acontece muitas vezes informalmente, quando as pessoas envolvidas no projeto sabem de projetos ou códigos semelhantes ao que é exigido. Elas os buscam, fazem as modificações necessárias e incorporam-nos a seus sistemas.

Esse reúso informal ocorre independentemente do processo de desenvolvimento que se use. No entanto, no século XXI, processos de desenvolvimento de software com foco no reúso de software existente tornaram-se amplamente usados. Abordagens orientadas a reúso dependem de uma ampla base de componentes reusáveis de software e de um *framework* de integração para a composição desses componentes. Em alguns casos, esses componentes são sistemas completos (COTS ou de prateleira), capazes de fornecer uma funcionalidade específica, como processamento de texto ou planilha.

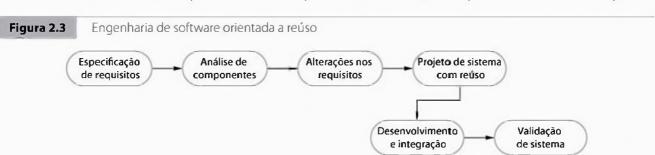
Um modelo de processo geral de desenvolvimento baseado no reúso está na Figura 2.3. Embora o estágio de especificação de requisitos iniciais e o estágio de validação sejam comparáveis a outros processos de software, os estágios intermediários em um processo orientado a reúso são diferentes. Esses estágios são:

1. *Análise de componentes.* Dada a especificação de requisitos, é feita uma busca por componentes para implementar essa especificação. Em geral, não há correspondência exata, e os componentes que podem ser usados apenas fornecem alguma funcionalidade necessária.
2. *Modificação de requisitos.* Durante esse estágio, os requisitos são analisados usando-se informações sobre os componentes que foram descobertos. Em seguida, estes serão modificados para refletir os componentes disponíveis. No caso de modificações impossíveis, a atividade de análise dos componentes pode ser reinserida na busca por soluções alternativas.
3. *Projeto do sistema com reúso.* Durante esse estágio, o *framework* do sistema é projetado ou algo existente é reusado. Os projetistas têm em mente os componentes que serão reusados e organizam o *framework* para reúso. Alguns softwares novos podem ser necessários, se componentes reusáveis não estiverem disponíveis.
4. *Desenvolvimento e integração.* Softwares que não podem ser adquiridos externamente são desenvolvidos, e os componentes e sistemas COTS são integrados para criar o novo sistema. A integração de sistemas, nesse modelo, pode ser parte do processo de desenvolvimento, em vez de uma atividade separada.

Existem três tipos de componentes de software que podem ser usados em um processo orientado a reúso:

1. *Web services* desenvolvidos de acordo com os padrões de serviço e que estão disponíveis para invocação remota.
2. Coleções de objetos que são desenvolvidas como um pacote a ser integrado com um *framework* de componentes, como .NET ou J2EE.
3. Sistemas de software *stand-alone* configurados para uso em um ambiente particular.

Engenharia de software orientada a reúso tem a vantagem óbvia de reduzir a quantidade de software a ser desenvolvido e, assim, reduzir os custos e riscos. Geralmente, também proporciona a entrega mais rápida do software. No entanto, compromisos com os requisitos são inevitáveis, e isso pode levar a um sistema que não

Figura 2.3 Engenharia de software orientada a reúso

atende às reais necessidades dos usuários. Além disso, algum controle sobre a evolução do sistema é perdido, pois as novas versões dos componentes reusáveis não estão sob o controle da organização que os está utilizando.

Reúso de software é um tema muito importante, ao qual dediquei vários capítulos na terceira parte deste livro. Questões gerais de reúso de software e reúso de COTS serão abordadas no Capítulo 16; a engenharia de software baseada em componentes, nos capítulos 17 e 18; e sistemas orientados a serviços, no Capítulo 19.

2.2 Atividades do processo

Processos reais de software são intercalados com sequências de atividades técnicas, de colaboração e de gerência, com o intuito de especificar, projetar, implementar e testar um sistema de software. Os desenvolvedores de software usam uma variedade de diferentes ferramentas de software em seu trabalho. As ferramentas são especialmente úteis para apoiar a edição de diferentes tipos de documentos e para gerenciar o imenso volume de informações detalhadas que é gerado em um projeto de grande porte.

As quatro atividades básicas do processo — especificação, desenvolvimento, validação e evolução — são organizadas de forma diferente conforme o processo de desenvolvimento. No modelo em cascata são organizadas em sequência, enquanto que no desenvolvimento incremental são intercaladas. A maneira como essas atividades serão feitas depende do tipo de software, das pessoas e das estruturas organizacionais envolvidas. Em *extreme programming*, por exemplo, as especificações estão escritas em cartões. Testes são executáveis e desenvolvidos antes do próprio programa. A evolução pode demandar reestruturação substancial do sistema ou refatoração.

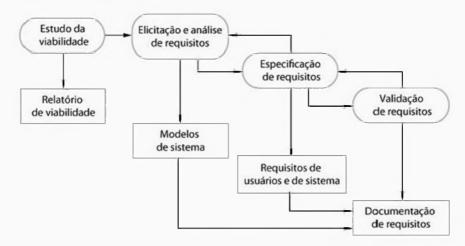
2.2.1 Especificação de software

Especificação de software ou engenharia de requisitos é o processo de compreensão e definição dos serviços requisitados do sistema e identificação de restrições relativas à operação e ao desenvolvimento do sistema. A engenharia de requisitos é um estágio particularmente crítico do processo de software, pois erros nessa fase inevitavelmente geram problemas no projeto e na implementação do sistema.

O processo de engenharia de requisitos (Figura 2.4) tem como objetivo produzir um documento de requisitos acordados que especifica um sistema que satisfaz os requisitos dos *stakeholders*. Requisitos são geralmente apresentados em dois níveis de detalhe. Os usuários finais e os clientes precisam de uma declaração de requisitos em alto nível; desenvolvedores de sistemas precisam de uma especificação mais detalhada do sistema.

Existem quatro atividades principais do processo de engenharia de requisitos:

1. *Estudo de viabilidade*. É feita uma estimativa acerca da possibilidade de se satisfazerem as necessidades do usuário identificado usando-se tecnologias atuais de software e hardware. O estudo considera se o sistema

Figura 2.4 Os requisitos da engenharia de processos

proposto será rentável a partir de um ponto de vista de negócio e se ele pode ser desenvolvido no âmbito das atuais restrições orçamentais. Um estudo de viabilidade deve ser relativamente barato e rápido. O resultado deve informar a decisão de avançar ou não, com uma análise mais detalhada.

2. *Elicitação e análise de requisitos*. Esse é o processo de derivação dos requisitos do sistema por meio da observação dos sistemas existentes, além de discussões com os potenciais usuários e compradores, análise de tarefas, entre outras etapas. Essa parte do processo pode envolver o desenvolvimento de um ou mais modelos de sistemas e protótipos, os quais nos ajudam a entender o sistema a ser especificado.
3. *Especificação de requisitos*. É a atividade de traduzir as informações obtidas durante a atividade de análise em um documento que defina um conjunto de requisitos. Dois tipos de requisitos podem ser incluídos nesse documento. Requisitos do usuário são declarações abstratas dos requisitos do sistema para o cliente e usuário final do sistema; requisitos de sistema são uma descrição mais detalhada da funcionalidade a ser provida.
4. *A validação de requisitos*. Essa atividade verifica os requisitos quanto a realismo, consistência e completude. Durante esse processo, os erros no documento de requisitos são inevitavelmente descobertos. Em seguida, o documento deve ser modificado para correção desses problemas.

Naturalmente, as atividades no processo de requisitos não são feitas em apenas uma sequência. A análise de requisitos continua durante a definição e especificação, e novos requisitos emergem durante o processo. Portanto, as atividades de análise, definição e especificação são intercaladas. Nos métodos ágeis, como *extreme programming*, os requisitos são desenvolvidos de forma incrementai, de acordo com as prioridades do usuário, e a elicitação de requisitos é feita pelos usuários que integram equipe de desenvolvimento.

2.2.2 Projeto e implementação de software

O estágio de implementação do desenvolvimento de software é o processo de conversão de uma especificação do sistema em um sistema executável. Sempre envolve processos de projeto e programação de software, mas, se for usada uma abordagem incremental para o desenvolvimento, também pode envolver o refinamento da especificação do software.

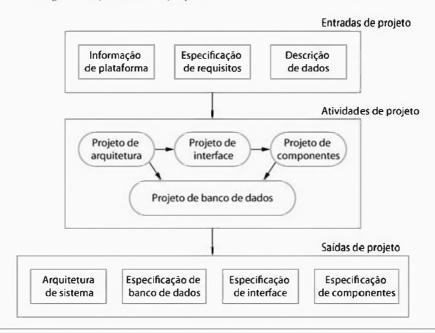
Um projeto de software é uma descrição da estrutura do software a ser implementado, dos modelos e estruturas de dados usados pelo sistema, das interfaces entre os componentes do sistema e, às vezes, dos algoritmos usados. Os projetistas não chegam a um projeto final imediatamente, mas desenvolvem-no de forma iterativa. Eles acrescentam formalidade e detalhes, enquanto desenvolvem seu projeto por meio de revisões constantes para correção de projetos anteriores.

A Figura 2.5 é um modelo abstrato de processo que mostra as entradas para o processo de projeto, suas atividades e os documentos produzidos como saídas dele. O diagrama sugere que os estágios do processo de projeto são sequenciais. Na realidade, as atividades do processo são intercaladas. *Feedback* de um estágio para outro e consequente retrabalho são inevitáveis em todos os processos.

A maioria dos softwares interage com outros sistemas de software, incluindo o sistema operacional, o banco de dados, o *middleware* e outros aplicativos. Estes formam a 'plataforma de software', o ambiente em que o software será executado. Informações sobre essa plataforma são entradas essenciais para o processo de projeto, pois os projetistas devem decidir a melhor forma de integrá-la ao ambiente do software. A especificação de requisitos é uma descrição da funcionalidade que o software deve oferecer, e seus requisitos de desempenho e confiança. Se o sistema for para processamento de dados existentes, a descrição desses dados poderia ser incluída na especificação da plataforma; caso contrário, a descrição dos dados deve ser uma entrada para o processo de projeto, para que a organização dos dados do sistema seja definida.

As atividades no processo de projeto podem variar, dependendo do tipo de sistema a ser desenvolvido. Por exemplo, sistemas de tempo real demandam projeto de *timing*, mas podem não incluir uma base de dados; nesse caso, não há um projeto de banco de dados envolvido. A Figura 2.5 mostra quatro atividades que podem ser parte do processo de projeto de sistemas de informação:

1. *Projeto de arquitetura*, no qual você pode identificar a estrutura geral do sistema, os componentes principais (algumas vezes, chamados subsistemas ou módulos), seus relacionamentos e como eles são distribuídos.
2. *Projeto de interface*, no qual você define as interfaces entre os componentes do sistema. Essa especificação de interface deve ser inequívoca. Com uma interface precisa, um componente pode ser usado de maneira que

Figura 2.5 Um modelo geral do processo de projeto

outros componentes não precisam saber como ele é implementado. Uma vez que as especificações de interface são acordadas, os componentes podem ser projetados e desenvolvidos simultaneamente.

3. *Projeto de componente*, no qual você toma cada componente do sistema e projeta seu funcionamento. Pode-se tratar de uma simples declaração da funcionalidade que se espera implementar, com o projeto específico para cada programador. Pode, também, ser uma lista de alterações a serem feitas em um componente reusável ou um modelo de projeto detalhado. O modelo de projeto pode ser usado para gerar automaticamente uma implementação.
4. *Projeto de banco de dados*, no qual você projeta as estruturas de dados do sistema e como eles devem ser representados em um banco de dados. Novamente, o trabalho aqui depende da existência de um banco de dados a ser reusado ou da criação de um novo banco de dados.

Essas atividades conduzem a um conjunto de saídas do projeto, que também é mostrado na Figura 2.5. O detalhe e a apresentação de cada uma varia consideravelmente. Para sistemas críticos, devem ser produzidos documentos detalhados de projeto, indicando as descrições precisas e exatas do sistema. Se uma abordagem dirigida a modelos é usada, essas saídas podem ser majoritariamente diagramas. Quando os métodos ágeis de desenvolvimento são usados, as saídas do processo de projeto podem não ser documentos de especificação separado, mas ser representadas no código do programa.

Métodos estruturados para projeto foram desenvolvidos nas décadas de 1970 e 1980, e foram os precursores da UML e do projeto orientado a objetos (BUDGEN, 2003). Eles estão relacionados com a produção de modelos gráficos do sistema e, em muitos casos, geram códigos automaticamente a partir desses modelos. Desenvolvimento dirigido a modelos (MDD, do inglês *model-driven development*) ou engenharia dirigida a modelos (SCHMIDT, 2006), em que os modelos de software são criados em diferentes níveis de abstração, é uma evolução dos métodos estruturados. Em MDD, há uma ênfase maior nos modelos de arquitetura com uma separação entre os modelos abstratos independentes de implementação e específicos de implementação. Os modelos são desenvolvidos em detalhes suficientes para que o sistema executável possa ser gerado a partir deles. Discuto essa abordagem para o desenvolvimento no Capítulo 5.

O desenvolvimento de um programa para implementar o sistema decorre naturalmente dos processos de projeto de sistema. Apesar de algumas classes de programa, como sistemas críticos de segurança, serem normalmente projetadas em detalhe antes de se iniciar qualquer implementação, é mais comum os estágios posteriores de projeto e desenvolvimento de programa serem intercalados. Ferramentas de desenvolvimento de software podem ser usadas para gerar um esqueleto de um programa a partir do projeto. Isso inclui o código para definir e implementar interfaces e, em muitos casos, o desenvolvedor precisa apenas acrescentar detalhes da operação de cada componente do programa.

Programação é uma atividade pessoal, não existe um processo geral a ser seguido. Alguns programadores começam com componentes que eles compreendem, desenvolvem-nos e depois passam para os componentes menos compreendidos. Outros preferem a abordagem oposta, deixando para o fim os componentes familiares, pois sabem como desenvolvê-los. Alguns desenvolvedores preferem definir os dados no início do processo e, em seguida, usam essa definição para dirigir o desenvolvimento do programa; outros deixam os dados não especificados durante o maior período de tempo possível.

Geralmente, os programadores fazem alguns testes do código que estão desenvolvendo, o que, muitas vezes, revela defeitos que devem ser retirados do programa. Isso é chamado *debugging*. Testes de defeitos e *debugging* são processos diferentes. Testes estabelecem a existência de defeitos; *debugging* diz respeito à localização e correção desses defeitos.

Quando você está realizando *debugging*, precisa gerar hipóteses sobre o comportamento observável do programa e, em seguida, testar essas hipóteses, na esperança de encontrar um defeito que tenha causado uma saída anormal. O teste das hipóteses pode envolver o rastreio manual do código do programa, bem como exigir novos casos de teste para localização do problema. Ferramentas interativas de depuração, que mostram os valores intermediários das variáveis do programa e uma lista das instruções executadas, podem ser usadas para apoiar o processo de depuração.

2.2.3 Validação de software

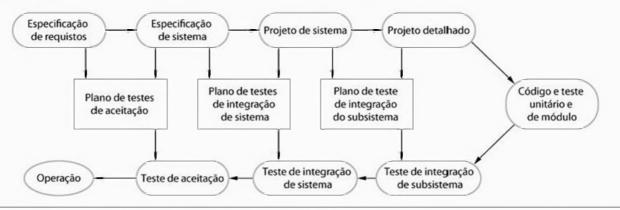
Validação de software ou, mais genericamente, verificação e validação (V&V), tem a intenção de mostrar que um software se adequa a suas especificações ao mesmo tempo que satisfaz as especificações do cliente do sistema. Teste de programa, em que o sistema é executado com dados de testes simulados, é a principal técnica de validação. A validação também pode envolver processos de verificação, como inspeções e revisões, em cada estágio do processo de software, desde a definição dos requisitos de usuários até o desenvolvimento do programa. Devido à predominância dos testes, a maior parte dos custos de validação incorre durante e após a implementação.

Com exceção de pequenos programas, sistemas não devem ser testados como uma unidade única e monolítica. A Figura 2.6 mostra um processo de teste, de três estágios, nos quais os componentes do sistema são testados, em seguida, o sistema integrado é testado e, finalmente, o sistema é testado com os dados do cliente. Idealmente, os defeitos de componentes são descobertos no início do processo, e os problemas de interface são encontrados quando o sistema é integrado. No entanto, quando os defeitos são descobertos, o programa deve ser depurado, e isso pode requerer que outros estágios do processo de testes sejam repetidos. Erros em componentes de programa podem vir à luz durante os testes de sistema. O proceso é, portanto, iterativo, com informações realimentadas de estágios posteriores para partes anteriores do processo.

Os estágios do processo de teste são:

1. *Testes de desenvolvimento*. Os componentes do sistema são testados pelas pessoas que o desenvolveram. Cada componente é testado de forma independente, separado dos outros. Os componentes podem ser entidades simples, como funções ou classes de objetos, ou podem ser agrupamentos coerentes dessas entidades. Ferramentas de automação de teste, como JUnit (MASSOL e HUSTED, 2003), que podem reexecutar testes de componentes quando as novas versões dos componentes são criadas, são comumente usadas.
2. *Testes de sistema*. Componentes do sistema são integrados para criar um sistema completo. Esse processo se preocupa em encontrar os erros resultantes das interações inesperadas entre componentes e problemas de interface do componente. Também visa mostrar que o sistema satisfaz seus requisitos funcionais e não funcionais, bem como testar as propriedades emergentes do sistema. Para sistemas de grande porte, esse pode ser

Figura 2.6 Estágios de testes

Figura 2.7 Fases de testes de um processo de software dirigido a planos

um processo multiestágios, no qual os componentes são integrados para formar subsistemas individualmente testados antes de serem integrados para formar o sistema final.

3. *Testes de aceitação*. Esse é o estágio final do processo de testes, antes que o sistema seja aceito para uso operacional. O sistema é testado com dados fornecidos pelo cliente, e não com dados advindos de testes simulados. O teste de aceitação pode revelar erros e omissões na definição dos requisitos do sistema, pois dados reais exercitam o sistema de formas diferentes dos dados de teste. Os testes de aceitação também podem revelar problemas de requisitos em que os recursos do sistema não atendam às necessidades do usuário ou o desempenho do sistema seja inaceitável.

Os processos de desenvolvimento de componentes e testes geralmente são intercalados. Os programadores criam seus próprios dados para testes e, incrementalmente, testam o código enquanto ele é desenvolvido. Essa é uma abordagem economicamente sensível, pois o programador conhece o componente e, portanto, é a melhor pessoa para gerar casos de teste.

Se uma abordagem incremental é usada para o desenvolvimento, cada incremento deve ser testado enquanto é desenvolvido — sendo que esses testes devem ser baseados nos requisitos para esse incremento. Em *extreme programming*, os testes são desenvolvidos paralelamente aos requisitos, antes de se iniciar o desenvolvimento, o que ajuda testadores e desenvolvedores a compreender os requisitos e garante o cumprimento dos prazos enquanto são criados os casos de teste.

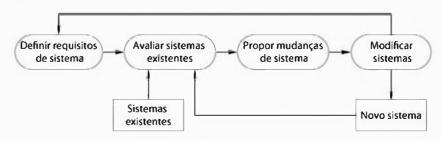
Quando um processo de software dirigido a planos é usado (por exemplo, para o desenvolvimento de sistemas críticos), o teste é impulsionado por um conjunto de planos de testes. Uma equipe independente de testadores trabalha a partir desses planos de teste pré-formulados, que foram desenvolvidos a partir das especificações e do projeto do sistema. A Figura 2.7 ilustra como os planos de teste são o elo entre as atividades de teste e de desenvolvimento. Esse modelo é, às vezes, chamado modelo V de desenvolvimento (gire a figura de lado para ver o V).

O teste de aceitação também pode ser chamado 'teste alfa'. Sistemas sob encomenda são desenvolvidos para um único cliente. O processo de testes-alfa continua até que o desenvolvedor do sistema e o cliente concordem que o sistema entregue é uma implementação aceitável dos requisitos.

Quando um sistema está pronto para ser comercializado como um produto de software, costuma-se usar um processo de testes denominado 'teste beta'. Este teste envolve a entrega de um sistema a um número de potenciais clientes que concordaram em usá-lo. Eles relatam problemas para os desenvolvedores dos sistemas. O produto é exposto para uso real, e erros que podem não ter sido antecipados pelos construtores do sistema são detectados. Após esse *feedback*, o sistema é modificado e liberado para outros testes-beta ou para venda em geral.

2.2.4 Evolução do software

A flexibilidade dos sistemas de software é uma das principais razões pelas quais os softwares vêm sendo, cada vez mais, incorporados em sistemas grandes e complexos. Uma vez que a decisão pela fabricação do hardware foi tomada, é muito caro fazer alterações em seu projeto. Entretanto, as mudanças no software podem ser feitas a qualquer momento durante ou após o desenvolvimento do sistema. Mesmo grandes mudanças são muito mais baratas do que as correspondentes alterações no hardware do sistema.

Figura 2.8 Evolução do sistema

Historicamente, sempre houve uma separação entre o processo de desenvolvimento e o de evolução do software (manutenção de software). As pessoas pensam no desenvolvimento de software como uma atividade criativa em que um sistema é desenvolvido a partir de um conceito inicial até um sistema funcional. Por outro lado, pensam na manutenção do software como maçante e desinteressante. Embora os custos de manutenção sejam frequntemente mais altos que os custos iniciais de desenvolvimento, os processos de manutenção são, em alguns casos, considerados menos desafiadores do que o desenvolvimento do software original.

Essa distinção entre o desenvolvimento e a manutenção é cada vez mais irrelevante. Poucos sistemas de software são completamente novos, e faz muito mais sentido ver o desenvolvimento e a manutenção como processos contínuos. Em vez de dois processos separados, é mais realista pensar na engenharia de software como um processo evolutivo (Figura 2.8), no qual o software é constantemente alterado durante seu período de vida em resposta às mudanças de requisitos e às necessidades do cliente.

2.3 Lidando com mudanças

A mudança é inevitável em todos os grandes projetos de software. Os requisitos do sistema mudam, ao mesmo tempo que o negócio que adquiriu o sistema responde a pressões externas e mudam as prioridades de gerenciamento. Com a disponibilidade de novas tecnologias, emergem novos projetos e possibilidades de implementação. Portanto, qualquer que seja o modelo do software de processo, é essencial que possa acomodar mudanças no software em desenvolvimento.

A mudança aumenta os custos de desenvolvimento de software, porque geralmente significa que o trabalho deve ser refeito. Isso é chamado retrabalho. Por exemplo, se os relacionamentos entre os requisitos do sistema foram analisados e novos requisitos foram identificados, alguma ou toda análise de requisitos deve ser repetida. Pode, então, ser necessário reprojetar o sistema de acordo com os novos requisitos, mudar qualquer programa que tenha sido desenvolvido e testar novamente o sistema.

Existem duas abordagens que podem ser adotadas para a redução de custos de retrabalho:

1. Prevenção de mudanças, em que o processo de software inclui atividades capazes de antecipar as mudanças possíveis antes que seja necessário qualquer retrabalho. Por exemplo, um protótipo de sistema pode ser desenvolvido para mostrar algumas características-chave do sistema para os clientes. Eles podem experimentar o protótipo e refinar seus requisitos antes de se comprometer com elevados custos de produção de software.
2. Tolerância a mudanças, em que o processo foi projetado para que as mudanças possam ser acomodadas a um custo relativamente baixo. Isso normalmente envolve alguma forma de desenvolvimento incremental. As alterações propostas podem ser aplicadas em incrementos que ainda não foram desenvolvidos. Se isso for impossível, então apenas um incremento (uma pequena parte do sistema) deve ser alterado para incorporar as mudanças.

Nesta seção, discuto duas maneiras de lidar com mudanças e mudanças nos requisitos do sistema. São elas:

1. Prototipação de sistema, em que uma versão do sistema ou de parte dele é desenvolvida rapidamente para verificar as necessidades do cliente e a viabilidade de algumas decisões de projeto. Esse processo previne mudanças, já que permite aos usuários experimentarem o sistema antes da entrega e, então, refinarem seus requisitos. O número de propostas de mudanças de requisitos a ser feito após a entrega é, portanto, suscetível de ser reduzido.

2. Entrega incremental, em que incrementos do sistema são entregues aos clientes para comentários e experimentação. Essa abordagem dá suporte tanto para a prevenção de mudanças quanto para a tolerância a mudanças. Também evita o comprometimento prematuro com requisitos para todo o sistema e permite que mudanças sejam incorporadas nos incrementos posteriores com um custo relativamente baixo.

A noção de refatoração, ou seja, melhoria da estrutura e organização de um programa, também é um importante mecanismo que suporta mudanças. Discuto esse assunto no Capítulo 3, que abrange métodos ágeis.

2 3.1 Prototipação

Um protótipo é uma versão inicial de um sistema de software, usado para demonstrar conceitos, experimentar opções de projeto e descobrir mais sobre o problema e suas possíveis soluções. O desenvolvimento rápido e iterativo do protótipo é essencial para que os custos sejam controlados e os *stakeholders* do sistema possam experimentá-lo no início do processo de software.

Um protótipo de software pode ser usado em um processo de desenvolvimento de software para ajudar a antecipar as mudanças que podem ser requisitadas:

1. No processo de engenharia de requisitos, um protótipo pode ajudar na elicitação e validação de requisitos de sistema.
2. No processo de projeto de sistema, um protótipo pode ser usado para estudar soluções específicas do software e para apoiar o projeto de interface de usuário.

Protótipos do sistema permitem aos usuários ver quão bem o sistema dá suporte a seu trabalho. Eles podem obter novas ideias para requisitos e encontrar pontos fortes e fracos do software; podem, então, propor novos requisitos do sistema. Além disso, o desenvolvimento do protótipo pode revelar erros e omissões nos requisitos propostos. A função descrita em uma especificação pode parecer útil e bem definida. No entanto, quando essa função é combinada com outras, os usuários muitas vezes percebem que sua visão inicial foi incorreta ou incompleta. A especificação do sistema pode então ser modificada para refletir o entendimento dos requisitos alterados.

Enquanto o sistema está em projeto, um protótipo do sistema pode ser usado para a realização de experimentos de projeto visando à verificação da viabilidade da proposta. Por exemplo, um projeto de banco de dados pode ser prototipado e testado para verificar se suporta de modo eficiente o acesso aos dados para as consultas mais comuns dos usuários. Prototipação também é uma parte essencial do processo de projeto da interface de usuário. Devido à natureza dinâmica de tais interfaces, descrições textuais e diagramas não são bons o suficiente para expressar seus requisitos. Portanto, a prototipação rápida com envolvimento do usuário final é a única maneira sensata de desenvolver interfaces gráficas de usuário para sistemas de software.

Um modelo de processo para desenvolvimento de protótipos é a Figura 2.9. Os objetivos da prototipação devem ser explicitados desde o início do processo. Estes podem ser o desenvolvimento de um sistema para prototipar a interface de usuário, o desenvolvimento de um sistema para validação dos requisitos funcionais do sistema ou o desenvolvimento de um sistema para demonstrar aos gerentes a viabilidade da aplicação. O mesmo protótipo não pode cumprir todos os objetivos. Se os objetivos não são declarados, a gerência ou os usuários finais podem não entender a função do protótipo. Consequentemente, eles podem não obter os benefícios que esperavam do desenvolvimento do protótipo.

O próximo estágio do processo é decidir o que colocar e, talvez mais importante ainda, o que deixar de fora do sistema de protótipo. Para reduzir os custos de prototipação e acelerar o cronograma de entrega, pode-se deixar alguma funcionalidade fora do protótipo. Você pode optar por relaxar os requisitos não funcionais, como tempo

Figura 2.9 O processo de desenvolvimento de protótipo

de resposta e utilização de memória. Gerenciamento e tratamento de erros podem ser ignorados, a menos que o objetivo do protótipo seja estabelecer uma interface de usuário. Padrões de confiabilidade e qualidade de programa podem ser reduzidos.

O estágio final do processo é a avaliação do protótipo. Durante esse estágio, provisões devem ser feitas para o treinamento do usuário, e os objetivos do protótipo devem ser usados para derivar um plano de avaliação. Os usuários necessitam de um tempo para se sentir confortáveis com um sistema novo e para se situarem em um padrão normal de uso. Uma vez que estejam usando o sistema normalmente, eles descobrem erros e omissões de requisitos.

Um problema geral com a prototipação é que o protótipo pode não ser necessariamente usado da mesma forma como o sistema final. O testador do protótipo pode não ser um usuário típico do sistema ou o tempo de treinamento durante a avaliação do protótipo pode ter sido insuficiente, por exemplo. Se o protótipo é lento, os avaliadores podem ajustar seu modo de trabalho e evitar os recursos do sistema que têm tempos de resposta lentos. Quando equipados com melhores respostas no sistema final, eles podem usá-lo de forma diferente.

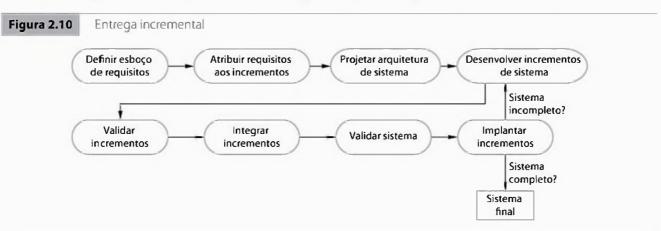
Às vezes, os desenvolvedores são pressionados pelos gerentes para entregar protótipos descartáveis, especialmente quando há atrasos na entrega da versão final do software. No entanto, isso costuma ser desaconselhável:

1. Pode ser impossível ajustar o protótipo para atender aos requisitos não funcionais, como requisitos de desempenho, proteção, robustez e confiabilidade, que foram ignorados durante o desenvolvimento do protótipo.
2. Mudanças rápidas durante o desenvolvimento inevitavelmente significam que o protótipo não está documentado. A única especificação de projeto é o código do protótipo. Para a manutenção a longo prazo, isso não é bom o suficiente.
3. As mudanças durante o desenvolvimento do protótipo provavelmente terão degradado a estrutura do sistema. O sistema será difícil e custoso de ser mantido.
4. Padrões de qualidade organizacional geralmente são relaxados para o desenvolvimento do protótipo.

Protótipos não precisam ser executáveis para serem úteis. Maquetes em papel da interface de usuário do sistema (RETTIG, 1994) podem ser eficazes em ajudar os usuários a refinar o projeto de interface e trabalhar por meio de cenários de uso. Estes são muito baratos de se desenvolver e podem ser construídos em poucos dias. Uma extensão dessa técnica é o protótipo Mágico de Oz, no qual apenas a interface de usuário é desenvolvida. Os usuários interagem com essa interface, mas suas solicitações são passadas para uma pessoa que os interpreta e produz a resposta adequada.

2.3.2 Entrega Incremental

Entrega incremental (Figura 2.10) é uma abordagem para desenvolvimento de software na qual alguns dos incrementos desenvolvidos são entregues ao cliente e implantados para uso em um ambiente operacional. Em um processo de entrega incremental os clientes identificam, em linhas gerais, os serviços a serem fornecidos pelo sistema. Eles identificam quais dos serviços são mais e menos importantes para eles. Uma série de incrementos de entrega são, então, definidos, com cada incremento proporcionando um subconjunto da funcionalidade do sistema. A atribuição de serviços aos incrementos depende da ordem de prioridade dos serviços — os serviços de mais alta prioridade são implementados e entregues em primeiro lugar.

Figura 2.10 Entrega incremental

Uma vez que os incrementos do sistema tenham sido identificados, os requisitos dos serviços a serem entregues no primeiro incremento são definidos em detalhes, e esse incremento é desenvolvido. Durante o desenvolvimento, podem ocorrer mais análises de requisitos para incrementos posteriores, mas mudanças nos requisitos do incremento atual não são aceitas.

Quando um incremento é concluído e entregue, os clientes podem colocá-lo em operação. Isso significa aceitar a entrega antecipada de uma parte da funcionalidade do sistema. Os clientes podem experimentar o sistema, e isso os ajuda a compreender suas necessidades para incrementos posteriores. Assim que novos incrementos são concluídos, eles são integrados aos incrementos existentes para que a funcionalidade do sistema melhore com cada incremento entregue.

A entrega incremental tem uma série de vantagens:

1. Os clientes podem usar os incrementos iniciais como protótipos e ganhar experiência, a qual informa seus requisitos para incrementos posteriores do sistema. Ao contrário de protótipos, trata-se, aqui, de partes do sistema real, ou seja, não existe a necessidade de reaprendizagem quando o sistema completo está disponível.
2. Os clientes não necessitam esperar até que todo o sistema seja entregue para obter ganhos a partir dele. O primeiro incremento satisfaz os requisitos mais críticos de maneira que eles possam usar o software imediatamente.
3. O processo mantém os benefícios do desenvolvimento incremental, o que deve facilitar a incorporação das mudanças no sistema.
4. Quanto maior a prioridade dos serviços entregues e, em seguida, incrementos integrados, os serviços mais importantes recebem a maioria dos testes. Isso significa que a probabilidade de os clientes encontrarem falhas de software nas partes mais importantes do sistema é menor.

No entanto, existem problemas com a entrega incremental:

1. A maioria dos sistemas exige um conjunto de recursos básicos, usados por diferentes partes do sistema. Como os requisitos não são definidos em detalhes até que um incremento possa ser implementado, pode ser difícil identificar recursos comuns, necessários a todos os incrementos.
2. O desenvolvimento iterativo também pode ser difícil quando um sistema substituto está sendo desenvolvido. Usuários querem toda a funcionalidade do sistema antigo e, muitas vezes, ficam relutantes em experimentar um novo sistema incompleto. Portanto, é difícil obter *feedbacks* úteis dos clientes.
3. A essência do processo iterativo é a especificação ser desenvolvida em conjunto com o software. Isso, contudo, causa conflitos com o modelo de compras de muitas organizações, em que a especificação completa do sistema é parte do contrato de desenvolvimento do sistema. Na abordagem incrementai, não há especificação completa do sistema até que o último incremento seja especificado, o que requer uma nova forma de contrato, à qual os grandes clientes, como agências governamentais, podem achar difícil de se adaptar.

Existem alguns tipos de sistema para os quais o desenvolvimento e a entrega incrementais não são a melhor abordagem. Esses sistemas são muito grandes, de modo que o desenvolvimento pode envolver equipes trabalhando em locais diferentes, além de alguns sistemas embutidos, em que o software depende do desenvolvimento de hardware, e de alguns sistemas críticos, em que todos os requisitos devem ser analisados na busca por interações capazes de comprometer a proteção ou a segurança do sistema.

Tais sistemas, naturalmente, sofrem com os mesmos problemas de requisitos incertos e mutáveis. Portanto, para resolver esses problemas e obter alguns dos benefícios do desenvolvimento incremental, pode ser usado um processo no qual um protótipo de sistema é desenvolvido de forma iterativa e usado como uma plataforma para experimentos com os requisitos e projeto do sistema. Com a experiência adquirida a partir do protótipo, requisitos definitivos podem ser, então, acordados.

2.3.3 Modelo espiral de Boehm

Um *framework* de processo de software dirigido a riscos (o modelo em espiral) foi proposto por Boehm (1988). Isso está na Figura 2.11. Aqui, o processo de software é representado como uma espiral, e não como uma sequência de atividades com alguns retornos de uma para outra. Cada volta na espiral representa uma fase do processo de software. Dessa forma, a volta mais interna pode preocupar-se com a viabilidade do sistema; o ciclo seguinte, com definição de requisitos; o seguinte, com o projeto do sistema, e assim por diante. O modelo em espiral combina prevenção e tolerância a mudanças, assume que mudanças são um resultado de riscos de projeto e inclui atividades explícitas de gerenciamento de riscos para sua redução.

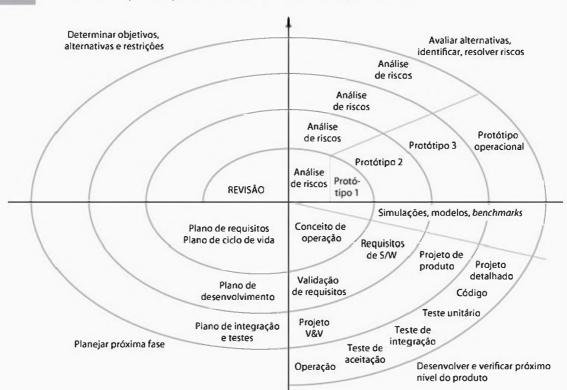



Figura 2.11 Modelo em espiral de processo de software de Boehm (©IEEE 1988)

Cada volta da espiral é dividida em quatro setores:

1. *Definição de objetivos*. Objetivos específicos para essa fase do projeto são definidos; restrições ao processo e ao produto são identificadas, e um plano de gerenciamento detalhado é elaborado; os riscos do projeto são identificados. Podem ser planejadas estratégias alternativas em função desses riscos.
2. *Avaliação e redução de riscos*. Para cada um dos riscos identificados do projeto, é feita uma análise detalhada. Medidas para redução do risco são tomadas. Por exemplo, se houver risco de os requisitos serem inadequados, um protótipo de sistema pode ser desenvolvido.
3. *Desenvolvimento e validação*. Após a avaliação dos riscos, é selecionado um modelo de desenvolvimento para o sistema. Por exemplo, a prototipação descartável pode ser a melhor abordagem de desenvolvimento de interface de usuário se os riscos forem dominantes. Se os riscos de segurança forem a principal consideração, o desenvolvimento baseado em transformações formais pode ser o processo mais adequado, e assim por diante. Se o principal risco identificado for a integração de subsistemas, o modelo em cascata pode ser a melhor opção.
4. *Planejamento*. O projeto é revisado, e uma decisão é tomada a respeito da continuidade do modelo com mais uma volta da espiral. Caso se decida pela continuidade, planos são elaborados para a próxima fase do projeto.

A principal diferença entre o modelo espiral e outros modelos de processo de software é seu reconhecimento explícito do risco. Um ciclo da espiral começa com a definição de objetivos, como desempenho e funcionalidade. Em seguida, são enumeradas formas alternativas de atingir tais objetivos e de lidar com as restrições de cada um deles. Cada alternativa é avaliada em função de cada objetivo, e as fontes de risco do projeto são identificadas. O próximo passo é resolver esses riscos por meio de atividades de coleta de informações, como análise mais detalhada, prototipação e simulação.

Após a avaliação dos riscos, algum desenvolvimento é efetivado, seguido por uma atividade de planejamento para a próxima fase do processo. De maneira informal dizemos que o risco significa, simplesmente, algo que pode dar errado. Por exemplo, se a intenção é usar uma nova linguagem de programação, um risco é o de os compiladores disponíveis não serem confiáveis ou não produzirem um código-objeto eficiente o bastante. Riscos levam a mudanças no software e problemas de projeto, como estouro de prazos e custos. Assim, o gerenciamento de riscos é uma atividade muito importante do projeto, constituindo uma das partes essenciais do gerenciamento de projetos, e será abordado no Capítulo 22.

2.4 Rational Unified Process (RUP)

O Rational Unified Process — RUP (KRUTCHEN, 2003) é um exemplo de modelo de processo moderno, derivado de trabalhos sobre a UML e o Unified Software Development Process associado (RUMBAUGH, et al., 1999; ARLOW e NEUSTADT, 2005). Incluí uma descrição aqui, pois é um bom exemplo de processo híbrido. Ele reúne elementos de todos os modelos de processo genéricos (Seção 2.1), ilustra boas práticas na especificação e no projeto (Seção 2.2) e apoia a prototipação e a entrega incremental (Seção 2.3).

O RUP reconhece que os modelos de processo convencionais apresentam uma visão única do processo. Em contrapartida, o RUP é normalmente descrito em três perspectivas:

1. Uma perspectiva dinâmica, que mostra as fases do modelo ao longo do tempo.
2. Uma perspectiva estática, que mostra as atividades relizadas no processo.
3. Uma perspectiva prática, que sugere boas práticas a serem usadas durante o processo.

A maioria das descrições do RUP tenta combinar as perspectivas estática e dinâmica em um único diagrama (KRUTCHEN, 2003). Por achar que essa tentativa torna o processo mais difícil de ser compreendido, uso descrições separadas de cada perspectiva.

O RUP é um modelo constituído de fases que identifica quatro fases distintas no processo de software. No entanto, ao contrário do modelo em cascata, no qual as fases são equalizadas com as atividades do processo, as fases do RUP são estreitamente relacionadas ao negócio, e não a assuntos técnicos. A Figura 2.12 mostra as fases do RUP. São elas:

1. *Concepção.* O objetivo da fase de concepção é estabelecer um *business case* para o sistema. Você deve identificar todas as entidades externas (pessoas e sistemas) que vão interagir com o sistema e definir as interações. Então, você deve usar essas informações para avaliar a contribuição do sistema para o negócio. Se essa contribuição for pequena, então o projeto poderá ser cancelado depois dessa fase.
2. *Elaboração.* As metas da fase de elaboração são desenvolver uma compreensão do problema dominante, estabelecer um *framework* da arquitetura para o sistema, desenvolver o plano do projeto e identificar os maiores riscos do projeto. No fim dessa fase, você deve ter um modelo de requisitos para o sistema, que pode ser um conjunto de casos de uso da UML, uma descrição da arquitetura ou um plano de desenvolvimento do software.
3. *Construção.* A fase de construção envolve projeto, programação e testes do sistema. Durante essa fase, as partes do sistema são desenvolvidas em paralelo e integradas. Na conclusão dessa fase, você deve ter um sistema de software já funcionando, bem como a documentação associada pronta para ser entregue aos usuários.
4. *Transição.* A fase final do RUP implica transferência do sistema da comunidade de desenvolvimento para a comunidade de usuários e em seu funcionamento em um ambiente real. Isso é ignorado na maioria dos modelos de processo de software, mas é, de fato, uma atividade cara e, às vezes, problemática. Na conclusão dessa fase, você deve ter um sistema de software documentado e funcionando corretamente em seu ambiente operacional.

No RUP, a iteração é apoiada de duas maneiras. Cada fase pode ser executada de forma iterativa com os resultados desenvolvidos de forma incremental. Além disso, todo o conjunto de fases também pode ser executado de forma incremental, como indicado pela seta curva de 'transição' para concepção, na Figura 2.12.

A visão estática do RUP prioriza as atividades que ocorrem durante o processo de desenvolvimento. Na descrição do RUP, essas são chamadas *workflows*. Existem seis *workflows* centrais, identificadas no processo, e três *workflows* de apoio. O RUP foi projetado em conjunto com a UML, assim, a descrição do *workflow* é orientada em

Figura 2.12 Fases no Rational Unified Process

torno de modelos associados à UML, como modelos de sequência, modelos de objetos etc. Os *workflows* centrais de engenharia e de apoio estão descritos na Tabela 2.1.

A vantagem de proporcionar visões estáticas e dinâmicas é que as fases do processo de desenvolvimento não estão associadas a *workflows* específicos. Ao menos em princípio, todos os *workflows* do RUP podem estar ativos em todas as fases do processo. Nas fases iniciais, provavelmente, maiores esforços serão empenhados em *workflows*, como modelagem de negócios e requisitos, e, nas fases posteriores, no teste e na implantação.

A perspectiva prática sobre o RUP descreve as boas práticas da engenharia de software que são recomendadas para uso no desenvolvimento de sistemas. Seis boas práticas fundamentais são recomendadas:

1. *Desenvolver software iterativamente*. Planejar os incrementos do sistema com base nas prioridades do cliente e desenvolver os recursos de alta prioridade no início do processo de desenvolvimento.
2. *Gerenciar os requisitos*. Documentar explicitamente os requisitos do cliente e acompanhar suas mudanças. Analisar o impacto das mudanças no sistema antes de aceitá-las.
3. *Usar arquiteturas baseadas em componentes*. Estruturar a arquitetura do sistema em componentes, conforme discutido anteriormente neste capítulo.
4. *Modelar o software visualmente*. Usar modelos gráficos da UML para apresentar visões estáticas e dinâmicas do software.
5. *Verificar a qualidade do software*. Assegurar que o software atenda aos padrões de qualidade organizacional.
6. *Controlar as mudanças do software*. Gerenciar as mudanças do software, usando um sistema de gerenciamento de mudanças e procedimentos e ferramentas de gerenciamento de configuração.

O RUP não é um processo adequado para todos os tipos de desenvolvimento, como, por exemplo, desenvolvimento de software embutido. No entanto, ele representa uma abordagem que potencialmente combina os três modelos de processo genéricos discutidos na Seção 2.1. As inovações mais importantes do RUP são a separação de fases e *workflows* e o reconhecimento de que a implantação de software em um ambiente do usuário é parte do processo. As fases são dinâmicas e têm metas. Os *workflows* são estáticos e são atividades técnicas que não são associadas a uma única fase, mas podem ser utilizadas durante todo o desenvolvimento para alcançar as metas específicas.

Tabela 2.1 *Workflows* estáticos no Rational Unified Process

WORKFLOW	**DESCRIÇÃO**
Modelagem de negócios	Os processos de negócio são modelados por meio de casos de uso de negócios.
Requisitos	Atores que interagem com o sistema são identificados e casos de uso são desenvolvidos para modelar os requisitos do sistema.
Análise e projeto	Um modelo de projeto é criado e documentado com modelos de arquitetura, modelos de componentes, modelos de objetos e modelos de sequência.
Implementação	Os componentes do sistema são implementados e estruturados em subsistemas de implementação. A geração automática de código a partir de modelos de projeto ajuda a acelerar esse processo.
Teste	O teste é um processo iterativo que é feito em conjunto com a implementação. O teste do sistema segue a conclusão da implementação.
Implantação	Um *release* do produto é criado, distribuído aos usuários e instalado em seu local de trabalho.
Gerenciamento de configuração e mudanças	Esse *workflow* de apoio gerencia as mudanças do sistema (veja o Capítulo 25).
Gerenciamento de projeto	Esse *workflow* de apoio gerencia o desenvolvimento do sistema (veja os capítulos 22 e 23).
Meio ambiente	Esse *workflow* está relacionado com a disponibilização de ferramentas apropriadas para a equipe de desenvolvimento de software.

PONTOS IMPORTANTES

- Os processos de software são as atividades envolvidas na produção de um sistema de software. Modelos de processos de software são representações abstratas desses processos.
- Modelos gerais de processo descrevem a organização dos processos de software. Exemplos desses modelos gerais incluem o modelo em cascata, o desenvolvimento incremental e o desenvolvimento orientado a reúso.
- Engenharia de requisitos é o processo de desenvolvimento de uma especificação de software. As especificações destinam-se a comunicar as necessidades de sistema dos clientes para os desenvolvedores do sistema.
- Processos de projeto e implementação estão relacionados com a transformação das especificações dos requisitos em um sistema de software executável. Métodos sistemáticos de projeto podem ser usados como parte dessa transformação.
- Validação de software é o processo de verificação de que o sistema está de acordo com sua especificação e satisfaz às necessidades reais dos usuários do sistema.
- Evolução de software ocorre quando se alteram os atuais sistemas de software para atender aos novos requisitos. As mudanças são contínuas, e o software deve evoluir para continuar útil.
- Processos devem incluir atividades para lidar com as mudanças. Podem envolver uma fase de prototipação, que ajuda a evitar más decisões sobre os requisitos e projeto. Processos podem ser estruturados para o desenvolvimento e a entrega iterativos, de forma que mudanças possam ser feitas sem afetar o sistema como um todo.
- O Rational Unified Process (RUP) é um moderno modelo genérico de processo, organizado em fases (concepção, elaboração, construção e transição), mas que separa as atividades (requisitos, análises, projeto etc.) dessas fases.

LEITURA COMPLEMENTAR

Managing Software Quality and Business Risk. Esse é essencialmente um livro sobre gerenciamento de software, mas que inclui um excelente capítulo (Capítulo 4) sobre os modelos de processo. (OULD, M. *Managing Software Quality and Business Risk*. John Wiley and Sons Ltd., 1999.)

"Process Models in Software Engineering". Essa é uma excelente visão geral de uma vasta gama de modelos de processo de engenharia de software que têm sido propostos. (SCACCHI, W. "Process Models in Software Engineering". In: MARCINIAK, J. J. (Orgs.). *Encyclopaedia of Software Engineering*. John Wiley and Sons, 2001.) Disponível em: <http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE- Encyc.pdf>.

The Rational Unified Process – An Introduction (*3rd Edition*). Esse é o livro sobre RUP mais lido no momento. Krutchen descreve bem o processo, mas eu gostaria de ter lido mais sobre as dificuldades práticas do uso do processo. (KRUTCHEN, P. *The Rational Unified Process – An Introduction*. 3.ed. Addison-Wesley, 2003.)

EXERCÍCIOS

2.1 Justificando sua resposta com base no tipo de sistema a ser desenvolvido, sugira o modelo genérico de processo de software mais adequado para ser usado como base para a gerência do desenvolvimento dos sistemas a seguir:

Um sistema para controlar o antibloqueio de frenagem de um carro.

Um sistema de realidade virtual para dar apoio à manutenção de software.

Um sistema de contabilidade para uma universidade, que substitua um sistema já existente.

Um sistema interativo de planejamento de viagens que ajude os usuários a planejar viagens com menor impacto ambiental.

2.2 Explique por que o desenvolvimento incremental é o método mais eficaz para o desenvolvimento de sistemas de software de negócios. Por que esse modelo é menos adequado para a engenharia de sistemas de tempo real?

2.3 Considere o modelo de processo baseado em reúso da Figura 2.3. Explique por que, nesse processo, é essencial ter duas atividades distintas de engenharia de requisitos.

2.4 Sugira por que é importante, no processo de engenharia de requisitos, fazer uma distinção entre desenvolvimento dos requisitos do usuário e desenvolvimento de requisitos de sistema.

2.5 Descreva as principais atividades do processo de projeto de software e as saídas dessas atividades. Usando um diagrama, mostre as possíveis relações entre as saídas dessas atividades.

2.6 Explique por que, em sistemas complexos, as mudanças são inevitáveis. Exemplifique as atividades de processo de software que ajudam a prever as mudanças e fazer com que o software seja desenvolvido mais tolerante a mudanças (desconsidere prototipação e entrega incremental).

2.7 Explique por que os sistemas desenvolvidos como protótipos normalmente não devem ser usados como sistemas de produção.

2.8 Explique por que o modelo em espiral de Boehm é um modelo adaptável, que apoia tanto as atividades de prevenção de mudanças quanto as de tolerância a mudanças. Na prática, esse modelo não tem sido amplamente usado. Sugira as possíveis razões para isso.

2.9 Quais são as vantagens de proporcionar visões estáticas e dinâmicas do processo de software, assim como no Rational Unified Process?

2.10 Historicamente, a introdução de tecnologia provocou mudanças profundas no mercado de trabalho e, pelo menos temporariamente, deixou muitas pessoas desempregadas. Discuta se a introdução da automação extensiva em processos pode vir a ter as mesmas consequências para os engenheiros de software. Se sua resposta for não, justifique. Se você acha que sim, que vai reduzir as oportunidades de emprego, é ética a resistência passiva ou ativa, pelos engenheiros afetados, à introdução dessa tecnologia?

REFERÊNCIAS

ARLOW, J.; NEUSTADT, I. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. 2.ed. Boston: Addison-Wesley, 2005.

BOEHM, B.; TURNER, R. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley, 2003.

BOEHM, B. W. "A Spiral Model of Software Development and Enhancement". *IEEE Computer*, v. 21, n. 5, 1988, p. 61-72.

BUDGEN, D. *Software Design*. 2.ed. Harlow, Reino Unido: Addison-Wesley, 2003.

KRUTCHEN, P. *The Rational Unified Process — An Introduction*. Reading, MA: Addison-Wesley, 2003.

MASSOL, V.; HUSTED, T. *JUnit in Action*. Greenwich, Conn.: Manning Publications Co., 2003.

RETTIG, M. "Practical Programmer: Prototyping for Tiny Fingers". *Comm. ACM*, v. 37, n. 4, 1994, p. 21-7.

ROYCE, W. W. "Managing the Development of Large Software Systems: Concepts and Techniques". *IEEE WESTCON*. Los Angeles, CA, 1970, p. 1-9.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified Software Development Process*. Reading, Mass: Addison-Wesley, 1999.

SCHMIDT, D. C. "Model-Driven Engineering". *IEEE Computer*, v. 39, n. 2, 2006, p. 25-31.

SCHNEIDER, S. *The B Method*. Houndmills, Reino Unido: Palgrave Macmillan, 2001.

WORDSWORTH, J. *Software Engineering with B*. Wokingham: Addison-Wesley, 1996.

CAPÍTULO

1 2 **3** 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Desenvolvimento ágil de software

Objetivos

O objetivo deste capítulo é apresentar os métodos ágeis de desenvolvimento de software. Ao terminar de ler este capítulo, você:

- compreenderá a lógica dos métodos ágeis de desenvolvimento de software, o manifesto ágil, e as diferenças entre desenvolvimento ágil e desenvolvimento dirigido a planos;
- conhecerá as práticas mais importantes da Extreme Programming e o modo como elas se relacionam com os princípios gerais dos métodos ágeis;
- compreenderá a abordagem Scrum para gerenciamento ágil de projetos;
- estará ciente das questões e problemas de escalamento de métodos ágeis de desenvolvimento para o desenvolvimento de sistemas de software de grande porte.

Conteúdo

Nos dias de hoje, as empresas operam em um ambiente global, com mudanças rápidas. Assim, precisam responder a novas oportunidades e novos mercados, a mudanças nas condições econômicas e ao surgimento de produtos e serviços concorrentes. Softwares fazem parte de quase todas as operações de negócios, assim, novos softwares são desenvolvidos rapidamente para obterem proveito de novas oportunidades e responder às pressões competitivas. O desenvolvimento e entrega rápidos são, portanto, o requisito mais crítico para o desenvolvimento de sistemas de software. Na verdade, muitas empresas estão dispostas a trocar a qualidade e o compromisso com requisitos do software por uma implantação mais rápida do software de que necessitam.

Essas empresas operam em um ambiente de mudanças rápidas, e por isso, muitas vezes, é praticamente impossível obter um conjunto completo de requisitos de software estável. Os requisitos iniciais inevitavelmente serão alterados, pois os clientes acham impossível prever como um sistema afetará as práticas de trabalho, como irá interagir com outros sistemas e quais operações do usuário devem ser automatizadas. Pode ser que os requisitos se tornem claros apenas após a entrega do sistema e à medida que os usuários ganhem experiência. Mesmo assim, devido a fatores externos, os requisitos são suscetíveis a mudanças rápidas e imprevisíveis. Por exemplo, quando for entregue, o software poderá estar desatualizado.

Processos de desenvolvimento de software que planejam especificar completamente os requisitos e, em seguida, projetar, construir e testar o sistema não estão adaptados ao desenvolvimento rápido de software. Com as mudanças nos requisitos ou a descoberta de problemas de requisitos, o projeto do sistema ou sua implementação precisa ser refeito ou

retestado. Como consequência, um processo convencional em cascata ou baseado em especificações costuma ser demorado, e o software final é entregue ao cliente bem depois do prazo acordado.

Para alguns tipos de software, como sistemas críticos de controle de segurança, em que uma análise completa do sistema é essencial, uma abordagem dirigida a planos é a melhor opção. No entanto, em um ambiente de negócio que se caracteriza por mudanças rápidas, isso pode causar problemas reais. Quando o software estiver disponível para uso, a razão original para sua aquisição pode ter mudado tão radicalmente que o software será efetivamente inútil. Portanto, para os sistemas de negócios, particularmente, os processos de desenvolvimento que se caracterizem por desenvolvimento e entrega rápidos de software são essenciais.

Há algum tempo já se reconhecia a necessidade de desenvolvimento rápido e de processos capazes de lidar com mudanças nos requisitos. Na década de 1980, a IBM introduziu o desenvolvimento incremental (MILLS et al., 1980). A introdução das linguagens de quarta geração, também em 1980, apoiou a ideia de desenvolvimento e entrega rápidos de software (MARTIN, 1981). No entanto, a ideia realmente decolou no final da década de 1990, com o desenvolvimento da noção de abordagens ágeis, como Metodologia de Desenvolvimento de Sistemas Dinâmicos (DSDM, do inglês *dynamic systems development method*) (STAPLETON, 1997), Scrum (SCHWABER e BEEDLE, 2001) e Extreme Programming (BECK, 1999; BECK, 2000).

Os processos de desenvolvimento rápido de software são concebidos para produzir, rapidamente, softwares úteis. O software não é desenvolvido como uma única unidade, mas como uma série de incrementos — cada incremento inclui uma nova funcionalidade do sistema. Embora existam muitas abordagens para o desenvolvimento rápido de software, elas compartilham algumas características fundamentais:

1. Os processos de especificação, projeto e implementação são intercalados. Não há especificação detalhada do sistema, e a documentação do projeto é minimizada ou gerada automaticamente pelo ambiente de programação usado para implementar o sistema. O documento de requisitos do usuário apenas define as características mais importantes do sistema.
2. O sistema é desenvolvido em uma série de versões. Os usuários finais e outros *stakeholders* do sistema são envolvidos na especificação e avaliação de cada versão. Eles podem propor alterações ao software e novos requisitos que devem ser implementados em uma versão posterior do sistema.
3. Interfaces de usuário do sistema são geralmente desenvolvidas com um sistema interativo de desenvolvimento que permite a criação rápida do projeto de interface por meio de desenho e posicionamento de ícones na interface. O sistema pode, então, gerar uma interface baseada na Web para um navegador ou uma interface para uma plataforma específica, como o Microsoft Windows.

Os métodos ágeis são métodos de desenvolvimento incremental em que os incrementos são pequenos e, normalmente, as novas versões do sistema são criadas e disponibilizadas aos clientes a cada duas ou três semanas. Elas envolvem os clientes no processo de desenvolvimento para obter *feedback* rápido sobre a evolução dos requisitos. Assim, minimiza-se a documentação, pois se utiliza mais a comunicação informal do que reuniões formais com documentos escritos.

3.1 Métodos ágeis

Na década de 1980 e início da de 1990, havia uma visão generalizada de que a melhor maneira para conseguir o melhor software era por meio de um planejamento cuidadoso do projeto, qualidade da segurança formalizada, do uso de métodos de análise e projeto apoiado por ferramentas CASE (*Computer-aided software engineering*) e do processo de desenvolvimento de software rigoroso e controlado. Essa percepção veio da comunidade de engenharia de software, responsável pelo desenvolvimento de sistemas de software grandes e duradouros, como sistemas aeroespaciais e de governo.

Esse software foi desenvolvido por grandes equipes que trabalham para diferentes empresas. Geralmente, as equipes eram dispersas geograficamente e trabalhavam com o software por longos períodos. Um exemplo desse tipo de software é o sistema de controle de uma aeronave moderna, que pode demorar até dez anos desde a especificação inicial até a implantação. Tais abordagens dirigidas a planos envolvem um *overhead* significativo no planejamento, projeto e documentação do sistema. Esse *overhead* se justifica quando o trabalho de várias equipes de desenvolvimento tem de ser coordenado, quando o sistema é um sistema crítico e quando muitas pessoas diferentes estão envolvidas na manutenção do software durante sua vida.

No entanto, quando essa abordagem pesada de desenvolvimento dirigido a planos é aplicada aos sistemas corporativos de pequeno e médio porte, o *overhead* envolvido é tão grande que domina o processo de desenvolvimento de software. Gasta-se mais tempo em análises de como o sistema deve ser desenvolvido do que no

desenvolvimento de programas e testes. Como os requisitos do sistema se alteram, o retrabalho é essencial, e, pelo menos em princípio, a especificação e o projeto devem mudar com o programa.

A insatisfação com essas abordagens pesadas da engenharia de software levou um grande número de desenvolvedores de software a proporem, na década de 1990, novos 'métodos ágeis'. Estes permitiram que a equipe de desenvolvimento focasse no software em si, e não em sua concepção e documentação. Métodos ágeis, universalmente, baseiam-se em uma abordagem incremental para a especificação, o desenvolvimento e a entrega do software. Eles são mais adequados ao desenvolvimento de aplicativos nos quais os requisitos de sistema mudam rapidamente durante o processo de desenvolvimento. Destinam-se a entregar o software rapidamente aos clientes, em funcionamento, e estes podem, em seguida, propor alterações e novos requisitos a serem incluídos nas iterações posteriores do sistema. Têm como objetivo reduzir a burocracia do processo, evitando qualquer trabalho de valor duvidoso de longo prazo e qualquer documentação que provavelmente nunca será usada.

A filosofia por trás dos métodos ágeis é refletida no manifesto ágil, que foi acordado por muitos dos principais desenvolvedores desses métodos. Esse manifesto afirma:

Estamos descobrindo melhores maneiras de desenvolver softwares, fazendo-o e ajudando outros a fazê-lo. Através desse trabalho, valorizamos mais:

Indivíduos e interações do que processos e ferramentas

Software em funcionamento do que documentação abrangente

Colaboração do cliente do que negociação de contrato

Respostas a mudanças do que seguir um plano

Ou seja, embora itens à direita sejam importantes, valorizamos mais os que estão à esquerda.

Provavelmente, o método ágil mais conhecido é a Extreme Programming (BECK, 1999; BECK, 2000), que descrevo adiante neste capítulo. Outras abordagens ágeis incluem Scrum (COHN, 2009; SCHWABER, 2004; SCHWABER e BEEDLE, 2001), Crystal (COCKBURN, 2001; COCKBURN, 2004), Desenvolvimento de Software Adaptativo (*Adaptative Software Development*), (HIGHSMITH, 2000), DSDM (STAPLETON, 1997; STAPLETON, 2003) e Desenvolvimento Dirigido a Características (*Feature Driven Development*), (PALMER e FELSING, 2002). O sucesso desses métodos levou a uma certa integração com métodos mais tradicionais de desenvolvimento baseados na modelagem do sistema, resultando no conceito de modelagem ágil (AMBLER e JEFFRIES, 2002) e instâncias ágeis do *Rational Unified Process* (LARMAN, 2002).

Embora esses métodos ágeis sejam todos baseados na noção de desenvolvimento e entrega incremental, eles propõem diferentes processos para alcançar tal objetivo. No entanto, compartilham um conjunto de princípios, com base no manifesto ágil, e por isso têm muito em comum. Esses princípios são mostrados na Tabela 3.1. Diferentes métodos ágeis instanciam esses princípios de maneiras diferentes, e eu não tenho espaço para discutir todos os métodos ágeis. Em vez disso, foco em dois dos mais usados: Extreme Programming (Seção 3.3) e Scrum (Seção 3.4).

Tabela 3.1 Os princípios dos métodos ágeis

Princípios	Descrição
Envolvimento do cliente	Os clientes devem estar intimamente envolvidos no processo de desenvolvimento. Seu papel é fornecer e priorizar novos requisitos do sistema e avaliar suas iterações.
Entrega incremental	O software é desenvolvido em incrementos com o cliente, especificando os requisitos para serem incluídos em cada um.
Pessoas, não processos	As habilidades da equipe de desenvolvimento devem ser reconhecidas e exploradas. Membros da equipe devem desenvolver suas próprias maneiras de trabalhar, sem processos prescritivos.
Aceitar as mudanças	Deve-se ter em mente que os requisitos do sistema vão mudar. Por isso, projete o sistema de maneira a acomodar essas mudanças.
Manter a simplicidade	Focalize a simplicidade, tanto do software a ser desenvolvido quanto do processo de desenvolvimento. Sempre que possível, trabalhe ativamente para eliminar a complexidade do sistema.

Métodos ágeis têm sido muito bem-sucedidos para alguns tipos de desenvolvimento de sistemas:

1. O desenvolvimento de produtos, em que uma empresa de software está desenvolvendo um produto pequeno ou médio para venda.
2. Desenvolvimento de sistema personalizado dentro de uma organização, em que existe um compromisso claro do cliente de se envolver no processo de desenvolvimento, e em que não há muitas regras e regulamentos externos que afetam o software.

Como discuto na seção final deste capítulo, o sucesso dos métodos ágeis indica que há um grande interesse em usar esses métodos para outros tipos de desenvolvimento de software. No entanto, devido a seu foco em pequenas equipes bem integradas, existem problemas em escalá-los para grandes sistemas. Existem também experiências de uso de abordagens ágeis para engenharia de sistemas críticos (DROBNA et al., 2004). No entanto, devido à necessidade de proteção, segurança e análise de confiança em sistemas críticos, exigem-se modificações significativas nos métodos ágeis antes que possam ser rotineiramente usados para a engenharia de sistemas críticos.

Na prática, os princípios básicos dos métodos ágeis são, por vezes, difíceis de se concretizar:

1. Embora a ideia de envolvimento do cliente no processo de desenvolvimento seja atraente, seu sucesso depende de um cliente disposto e capaz de passar o tempo com a equipe de desenvolvimento, e que possa representar todos os *stakeholders* do sistema. Frequentemente, os representantes dos clientes estão sujeitos a diversas pressões e não podem participar plenamente do desenvolvimento de software.
2. Membros individuais da equipe podem não ter personalidade adequada para o intenso envolvimento que é típico dos métodos ágeis e, portanto, não interagem bem com outros membros da equipe.
3. Priorizar as mudanças pode ser extremamente difícil, especialmente em sistemas nos quais existem muitos *stakeholders*. Normalmente, cada *stakeholder* dá prioridades diferentes para mudanças diferentes.
4. Manter a simplicidade exige um trabalho extra. Sob a pressão de cronogramas de entrega, os membros da equipe podem não ter tempo para fazer as simplificações desejáveis.
5. Muitas organizações, principalmente as grandes empresas, passaram anos mudando sua cultura para que os processos fossem definidos e seguidos. É difícil para eles mudar de um modelo de trabalho em que os processos são informais e definidos pelas equipes de desenvolvimento.

Outro problema não técnico — que é um problema geral do desenvolvimento e da entrega incremental — ocorre quando o cliente do sistema usa uma organização externa para desenvolver o sistema. O documento de requisitos do software é normalmente parte do contrato entre o cliente e o fornecedor. Como a especificação incremental é inerente aos métodos ágeis, escrever contratos para esse tipo de desenvolvimento pode ser difícil.

Consequentemente, os métodos ágeis têm de contar com contratos nos quais o cliente paga pelo tempo necessário para o desenvolvimento do sistema, e não pelo desenvolvimento de um determinado conjunto de requisitos. Enquanto tudo vai bem, isso beneficia o cliente e o desenvolvedor. No entanto, se surgirem problemas, poderão acontecer disputas nas quais fica difícil definir quem é culpado e quem deve pagar pelo tempo extra, bem como os recursos necessários para a solução dos problemas.

A maioria dos livros e artigos que descrevem os métodos ágeis e experiências com eles fala sobre o uso desses métodos para o desenvolvimento de novos sistemas. No entanto, como explico no Capítulo 9, um enorme esforço da engenharia de software é dedicado à manutenção e à evolução de sistemas de software já existentes. Existe apenas um pequeno número de relatos de experiências com o uso de métodos ágeis na manutenção de software (POOLE e HUISMAN, 2001). Há duas questões que devem ser consideradas ao tratarmos de métodos ágeis e manutenção:

1. É possível fazer manutenção dos sistemas desenvolvidos em uma abordagem ágil, dada a ênfase do processo de desenvolvimento em minimização da documentação formal?
2. Os métodos ágeis podem, efetivamente, ser usados para a evolução de um sistema em resposta às solicitações de mudança do cliente?

Supostamente, a documentação formal deve descrever o sistema e, assim, tornar sua compreensão mais fácil para as pessoas que fazem as mudanças. Porém, na prática, a documentação formal nem sempre é atualizada, e, portanto, não reflete exatamente o código do programa. Por essa razão, os entusiastas de métodos ágeis argumentam que é um desperdício de tempo criar essa documentação e que a chave para a implementação de software manutenível é a produção de códigos de alta qualidade, legíveis. Práticas ágeis, portanto, enfatizam a importância de se escrever códigos bem-estruturados e investir na melhoria do código. Portanto, a falta de documentação não deve ser um problema na manutenção dos sistemas desenvolvidos por meio de uma abordagem ágil.

No entanto, minha experiência em manutenção de sistemas sugere que o documento-chave é o documento de requisitos do sistema, que informa ao engenheiro de software o que o sistema deve fazer. Sem esse conhecimento, é difícil avaliar o impacto das mudanças propostas. Muitos métodos ágeis coletam requisitos informalmente, de forma incremental, e não desenvolvem um documento coerente de requisitos. Nesse sentido, o uso de métodos ágeis pode tornar mais difícil e cara a manutenção posterior do sistema.

Práticas ágeis, usadas no processo de manutenção em si, provavelmente são mais eficazes, independente de ter sido usada uma abordagem ágil para o desenvolvimento do sistema. Entrega incremental, projeto para mudanças e manutenção da simplicidade — tudo isso faz sentido quando o software está sendo alterado. Na verdade, você pode pensar em um processo ágil de desenvolvimento como um processo de evolução do software.

No entanto, a principal dificuldade após a entrega do software é manter o envolvimento dos clientes no processo. Apesar de, durante o desenvolvimento do sistema, um cliente poder justificar o envolvimento de um representante em tempo integral, isso é menos provável durante a manutenção, período em que as mudanças não são contínuas. Representantes do cliente são propensos a perder o interesse no sistema. Portanto, para criar novos requisitos do sistema podem ser necessários mecanismos alternativos, como as propostas de mudanças discutidas no Capítulo 25.

O outro problema que pode surgir está relacionado à continuidade da equipe de desenvolvimento. Métodos ágeis dependem de os membros da equipe compreenderem aspectos do sistema sem consultar a documentação. Se uma equipe de desenvolvimento ágil é alterada, esse conhecimento implícito é perdido, e é difícil para os novos membros da equipe construir o mesmo entendimento do sistema e seus componentes.

Os defensores dos métodos ágeis foram evangelizadores na promoção do uso desses métodos e tenderam a negligenciar suas deficiências. Isso provocou uma resposta igualmente extrema, que, em minha opinião, exagera os problemas dessa abordagem (STEPHENS e ROSENBERG, 2003). Críticos mais fundamentados, como DeMarco e Boehm (2002), destacam as vantagens e desvantagens dos métodos ágeis. Eles propõem uma abordagem híbrida, na qual os métodos ágeis incorporam algumas técnicas do desenvolvimento dirigido a planos, que pode ser o melhor caminho a seguir.

3.2 Desenvolvimento ágil e dirigido a planos

Abordagens ágeis de desenvolvimento de software consideram o projeto e a implementação como atividades centrais no processo de software. Eles incorporam outras atividades, como elicitação de requisitos e testes no projeto e na implementação. Em contrapartida, uma abordagem de engenharia de software dirigida a planos identifica estágios distintos do processo de software com saídas associadas a cada estágio. As saídas de um estágio são usadas como base para o planejamento da atividade do processo a seguir. A Figura 3.1 mostra as distinções entre as abordagens dirigidas a planos e ágil para a especificação do sistema.

Em uma abordagem dirigida a planos, ocorrem iterações no âmbito das atividades com documentos formais, usados para estabelecer a comunicação entre os estágios do processo. Por exemplo, os requisitos vão evoluir e, finalmente, será produzida uma especificação de requisitos. Essa é, então, uma entrada para o processo de projeto e implementação. Em uma abordagem ágil, iterações ocorrem em todas as atividades. Portanto, os requisitos e o projeto são desenvolvidos em conjunto, e não separadamente.

Um processo de software dirigido a planos pode apoiar o desenvolvimento e a entrega incremental. É perfeitamente possível alocar requisitos e planejar as fases de projeto e desenvolvimento como uma série de incrementos. Um processo ágil não é, inevitavelmente, focado no código, e pode produzir alguma documentação de projeto. Como vou discutir na seção seguinte, a equipe de desenvolvimento ágil pode decidir incluir um '*spike*' de documentação, no qual, em vez de produzir uma nova versão de um sistema, a equipe produz documentação do sistema.

Na verdade, a maioria dos projetos de software inclui práticas das abordagens dirigidas a planos e ágil. Para optar por um equilíbrio entre as abordagens, você precisa responder a uma série de questões técnicas, humanas e organizacionais:

1. É importante ter uma especificação e um projeto muito detalhados antes de passar para a implementação? Se sim, você provavelmente necessita usar uma abordagem dirigida a planos.
2. É realista uma estratégia de entrega incremental em que você entrega o software aos clientes e rapidamente obtém um *feedback*? Em caso afirmativo, considere o uso de métodos ágeis.

Figura 3.1 Especificações dirigida a planos e ágil

Desenvolvimento baseado em planos

Engenharia de requisitos

Especificação de requisitos

Projeto e implementação

Solicitação de mudança de requisitos

Desenvolvimento ágil

Engenharia de requisitos

Projeto e implementação

3. Quão grande é o sistema que está em desenvolvimento? Os métodos ágeis são mais eficazes quando o sistema pode ser desenvolvido com uma pequena equipe colocalizada capaz de se comunicar de maneira informal. Isso pode não ser possível para sistemas de grande porte que exigem equipes de desenvolvimento maiores — nesse caso, uma abordagem dirigida a planos pode ter de ser usada.
4. Que tipo de sistema está sendo desenvolvido? Sistemas que exigem uma análise profunda antes da implementação (por exemplo, sistema de tempo real com requisitos de tempo complexos) geralmente demandam um projeto bastante detalhado para atender a essa análise. Nessas circunstâncias, uma abordagem dirigida a planos pode ser a melhor opção.
5. Qual é o tempo de vida esperado do sistema? Sistemas de vida-longa podem exigir mais da documentação de projeto, a fim de comunicar para a equipe de apoio as intenções originais dos desenvolvedores do sistema. No entanto, os defensores dos métodos ágeis argumentam, corretamente, que a documentação não é constantemente atualizada e não é de muita utilidade para a manutenção do sistema a longo prazo.
6. Que tecnologias estão disponíveis para apoiar o desenvolvimento do sistema? Métodos ágeis frequentemente contam com boas ferramentas para manter o controle de um projeto em desenvolvimento. Se você está desenvolvendo um sistema utilizando um IDE (ambiente integrado de desenvolvimento, do inglês *integrated development environment*) que não tem boas ferramentas para visualização e análise do programa, pode haver necessidade de mais documentação do projeto.
7. Como é organizada a equipe de desenvolvimento? Se está distribuída, ou se parte do desenvolvimento está sendo terceirizado, então pode ser necessário o desenvolvimento de documentos de projeto para a comunicação entre as equipes de desenvolvimento. Pode ser necessário planejar com antecedência quais serão esses documentos.
8. Existem questões culturais que podem afetar o desenvolvimento do sistema? Organizações tradicionais de engenharia têm uma cultura de desenvolvimento baseado em planos, pois essa é a norma na engenharia. Geralmente, isso requer extensa documentação de projeto, no lugar do conhecimento informal, usado em processos ágeis.
9. Quão bons são os projetistas e programadores na equipe de desenvolvimento? Às vezes, argumenta-se que os métodos ágeis exigem níveis mais altos de habilidade do que as abordagens dirigidas a planos, em que os programadores simplesmente traduzem um projeto detalhado em um código. Se você tem uma equipe com níveis de habilidade relativamente baixos, pode precisar usar as melhores pessoas para desenvolver o projeto, juntamente com outros, responsáveis pela programação.
10. O sistema é sujeito à regulamentação externa? Se um sistema tem de ser aprovado por um regulador externo (por exemplo, a FAA [Autoridade Federal de Aviação, do inglês *Federal Aviation Authority*] aprova os softwares

críticos para a operação de uma aeronave), então, provavelmente, será obrigatória a produção de uma documentação detalhada como parte da documentação de segurança do sistema.

Na realidade, a questão sobre rotular o projeto como dirigido a planos ou ágil não é muito importante. Em última análise, a principal preocupação dos compradores de um sistema de software é se estão comprando um sistema de software executável que atenda às suas necessidades e faça coisas úteis para o usuário individual ou para a organização. Na prática, muitas empresas que afirmam ter usado métodos ágeis adotaram algumas práticas ágeis e integraram-nas em seus processos dirigidos a planos.

3.3 Extreme Programming

Extreme Programming (XP) é talvez o mais conhecido e mais utilizado dos métodos ágeis. O nome foi cunhado por Beck (2000), pois a abordagem foi desenvolvida para impulsionar práticas reconhecidamente boas, como o desenvolvimento iterativo, a níveis 'extremos'. Por exemplo, em XP, várias novas versões de um sistema podem ser desenvolvidas, integradas e testadas em um único dia por programadores diferentes.

Em Extreme Programming, os requisitos são expressos como cenários (chamados de histórias do usuário), que são implementados diretamente como uma série de tarefas. Os programadores trabalham em pares e desenvolvem testes para cada tarefa antes de escreverem o código. Quando o novo código é integrado ao sistema, todos os testes devem ser executados com sucesso. Há um curto intervalo entre os *releases* do sistema. A Figura 3.2 ilustra o processo XP para a produção de um incremento do sistema que está sendo desenvolvido.

Extreme Programming envolve uma série de práticas que refletem os princípios dos métodos ágeis (elas estão resumidas na Tabela 3.2):

1. O desenvolvimento incremental é sustentado por meio de pequenos e frequentes releases do sistema. Os requisitos são baseados em cenários ou em simples histórias de clientes, usadas como base para decidir a funcionalidade que deve ser incluída em um incremento do sistema.
2. O envolvimento do cliente é sustentado por meio do engajamento contínuo do cliente com a equipe de desenvolvimento. O representante do cliente participa do desenvolvimento e é responsável por definir os testes de aceitação para o sistema.
3. Pessoas — não processos — são sustentadas por meio de programação em pares, propriedade coletiva do código do sistema e um processo de desenvolvimento sustentável que não envolve horas de trabalho excessivamente longas.
4. As mudanças são aceitas por meio de *releases* contínuos para os clientes, do desenvolvimento *test-first*, da refatoração para evitar a degeneração do código e integração contínua de nova funcionalidade.
5. A manutenção da simplicidade é feita por meio da refatoração constante que melhora a qualidade do código, bem como por meio de projetos simples que não antecipam desnecessariamente futuras mudanças no sistema.

Em um processo XP, os clientes estão intimamente envolvidos na especificação e priorização dos requisitos do sistema. Os requisitos não estão especificados como uma lista de funções requeridas do sistema. Pelo contrário, o cliente do sistema é parte da equipe de desenvolvimento e discute cenários com outros membros da equipe. Juntos, eles desenvolvem um 'cartão de história', englobando as necessidades do cliente. A equipe de

Figura 3.2 O ciclo de um *release* em Extreme Programming

Tabela 3.2 Práticas de Extreme Programming

Princípio ou prática	**Descrição**
Planejamento incremental	Os requisitos são gravados em cartões de história e as histórias que serão incluídas em um release são determinadas pelo tempo disponível e sua relativa prioridade. Os desenvolvedores dividem essas histórias em 'Tarefas'. Veja os quadros 3.1 e 3.2.
Pequenos *releases*	Em primeiro lugar, desenvolve-se um conjunto mínimo de funcionalidades útil, que fornece o valor do negócio. *Releases* do sistema são frequentes e gradualmente adicionam funcionalidade ao primeiro *release*.
Projeto simples	Cada projeto é realizado para atender às necessidades atuais, e nada mais.
Desenvolvimento *test-first*	Um *framework* de testes iniciais automatizados é usado para escrever os testes para uma nova funcionalidade antes que a funcionalidade em si seja implementada.
Refatoração	Todos os desenvolvedores devem refatorar o código continuamente assim que encontrarem melhorias de código. Isso mantém o código simples e manutenível.
Programação em pares	Os desenvolvedores trabalham em pares, verificando o trabalho dos outros e prestando apoio para um bom trabalho sempre.
Propriedade coletiva	Os pares de desenvolvedores trabalham em todas as áreas do sistema, de modo que não se desenvolvam ilhas de *expertise*. Todos os conhecimentos e todos os desenvolvedores assumem responsabilidade por todo o código. Qualquer um pode mudar qualquer coisa.
Integração contínua	Assim que o trabalho em uma tarefa é concluído, ele é integrado ao sistema como um todo. Após essa integração, todos os testes de unidade do sistema devem passar.
Ritmo sustentável	Grandes quantidades de horas-extra não são consideradas aceitáveis, pois o resultado final, muitas vezes, é a redução da qualidade do código e da produtividade a médio prazo.
Cliente no local	Um representante do usuário final do sistema (o cliente) deve estar disponível todo o tempo à equipe de XP. Em um processo de Extreme Programming, o cliente é um membro da equipe de desenvolvimento e é responsável por levar a ela os requisitos de sistema para implementação.

desenvolvimento, então, tenta implementar esse cenário em um *release* futuro do software. O Quadro 3.1 mostra um exemplo de um cartão de história para o gerenciamento do sistema de cuidado da saúde mental de pacientes. Essa é uma breve descrição de um cenário para a prescrição de medicamentos a um paciente.

Quadro 3.1 Uma história de prescrição de medicamentos

Prescrição de medicamentos
Kate é uma médica que deseja prescrever medicamentos para um paciente de uma clínica. O prontuário do paciente já está sendo exibido em seu computador, assim, ela clica o campo 'medicação' e pode selecionar 'medicação atual', 'nova medicação', ou 'formulário'. Se ela selecionar 'medicação atual', o sistema pede que ela verifique a dose. Se ela quiser mudar a dose, ela altera esta e em seguida, confirma a prescrição. Se ela escolher 'nova medicação', o sistema assume que ela sabe qual medicação receitar. Ela digita as primeiras letras do nome do medicamento. O sistema exibe uma lista de possíveis fármacos que começam com essas letras. Ela escolhe a medicação requerida e o sistema responde, pedindo-lhe para verificar se o medicamento selecionado está correto. Ela insere a dose e, em seguida, confirma a prescrição. Se ela escolhe 'formulário', o sistema exibe uma caixa de busca para o formulário aprovado. Ela pode, então, procurar pelo medicamento requerido. Ela seleciona um medicamento e é solicitado que verifique se a medicação está correta. Ela insere a dose e, em seguida, confirma a prescrição. O sistema sempre verifica se a dose está dentro da faixa permitida. Caso não esteja, Kate é convidada a alterar a dose. Após Kate confirmar a prescrição, esta será exibida para verificação. Ela pode escolher 'OK' ou 'Alterar'. Se clicar em 'OK', a prescrição fica gravada nos bancos de dados da auditoria. Se ela clicar em 'Alterar', reinicia o processo de 'Prescrição de Medicamentos'.

Os cartões de história são as principais entradas para o processo de planejamento em XP ou 'jogo de planejamento'. Uma vez que tenham sido desenvolvidos, a equipe de desenvolvimento os divide em tarefas (Quadro 3.2) e estima o esforço e os recursos necessários para a realização de cada tarefa. Esse processo geralmente envolve discussões com o cliente para refinamento dos requisitos. O cliente, então, prioriza as histórias para implementação, escolhendo aquelas que podem ser usadas imediatamente para oferecer apoio aos negócios. A intenção é identificar funcionalidade útil que possa ser implementada em cerca de duas semanas, quando o próximo *release* do sistema é disponibilizado para o cliente.

Claro que, como os requisitos mudam, as histórias não implementadas mudam ou podem ser descartadas. Se houver necessidade de mudanças em um sistema que já tenha sido entregue, novos cartões de história são desenvolvidos e, mais uma vez, o cliente decide se essas mudanças devem ter prioridade sobre a nova funcionalidade.

Às vezes, durante o jogo de planejamento, emergem questões que não podem ser facilmente respondidas, tornando necessário algum trabalho adicional para explorar possíveis soluções. A equipe pode fazer algum protótipo ou desenvolvimento-teste para entender o problema e a solução. Em termos XP, isso é um '*spike*', um incremento em que nenhum tipo de programação é realizado. Também pode haver '*spikes*' de projeto da arquitetura do sistema ou para desenvolver a documentação do sistema.

Extreme Programming leva uma abordagem 'extrema' para o desenvolvimento incremental. Novas versões do software podem ser construídas várias vezes por dia e *releases* são entregues aos clientes a cada duas semanas, aproximadamente. Prazos de *releases* nunca são desrespeitados; se houver problemas de desenvolvimento, o cliente é consultado, e a funcionalidade é removida do *release* planejado.

Quando um programador constrói o sistema para criar uma nova versão, deve executar todos os testes automatizados existentes, bem como os testes para a nova funcionalidade. A nova construção do software só é aceita se todos os testes forem executados com êxito. Esta se torna, então, a base para a próxima iteração do sistema.

Um preceito fundamental da engenharia de software tradicional é que você deve projetar para mudar. Ou seja, você deve antecipar futuras alterações do software e projetá-lo para que essas mudanças possam ser facilmente implementadas. O Extreme Programming, no entanto, descartou esse princípio com base na concepção de que muitas vezes a mudança é um esforço desperdiçado. Não vale a pena perder tempo adicionando generalidades a um programa para lidar com mudanças. Frequentemente, mudanças previstas não se materializam e solicitações por mudanças completamente diferentes podem ser feitas. Portanto, a abordagem XP aceita que as mudanças acontecerão e reorganizarão o software quando essas mudanças realmente acontecerem.

Um problema geral com o desenvolvimento incremental é que ele tende a degradar a estrutura do software. Desse modo, as mudanças para o software tornam-se cada vez mais difíceis de serem implementadas. Essencialmente, o desenvolvimento prossegue, encontrando soluções para os problemas, mas o resultado final frequentemente é a duplicação do código; partes do software são reusadas de maneira inadequada, e a estrutura global do código degrada-se quando ele é adicionado ao sistema.

Quadro 3.2 Exemplos de cartões de tarefa para a prescrição de medicamentos

Tarefa 1: Alterar dose de medicamentos prescritos

Tarefa 2: Seleção de formulário

Tarefa 3: Verificação de dose

A verificação da dose é uma precaução de segurança para verificar se o médico não receitou uma dose perigosamente pequena ou grande.

Usando o ID do formulário para o nome do medicamento genérico, procure o formulário e obtenha a dose mínima e máxima recomendada.

Verifique a dose mínima e máxima prescrita. Caso esteja fora da faixa, emita uma mensagem de erro dizendo que a dose está muito alta ou muito baixa.

Caso esteja dentro da faixa, habilite o botão 'Confirmar'.

Extreme Programming aborda esse problema, sugerindo que o software deve ser constantemente refatorado. Isso significa que a equipe de programação deve estar focada em possíveis melhorias para o software e em implementação imediata destas. Quando um membro da equipe percebe que o código pode ser melhorado, essas melhorias são feitas, mesmo quando não existe necessidade imediata destas. Exemplos de refatoração incluem a reorganização da hierarquia de classes para eliminação de código duplicado, a arrumação e renomeação de atributos e métodos, bem como a substituição do código com as chamadas para métodos definidos em uma biblioteca de programas. Ambientes de desenvolvimento de programas, como o Eclipse (CARLSON, 2005), incluem ferramentas de refatoração que simplificam o processo de encontrar dependências entre as seções do código e fazer as modificações no código global.

Em princípio, portanto, o software deve ser sempre fácil de compreender e mudar à medida que novas histórias sejam implementadas. Na prática, isso nem sempre ocorre. Em alguns casos, a pressão pelo desenvolvimento significa que a refatoração será postergada, porque a maior parte do tempo é dedicada à implementação de nova funcionalidade. Algumas novas características e mudanças não podem ser facilmente acomodadas por refatoração do nível do código, e exigem modificações da arquitetura do sistema.

Na prática, muitas empresas que adotaram XP não usam todas as práticas da Extreme Programming listadas na Tabela 3.2. Elas escolhem de acordo com sua organização. Por exemplo, algumas empresas consideram útil a programação em pares, outras preferem a programação individual e revisões. Para acomodar os diferentes níveis de habilidade, alguns programadores não fazem refatoração em partes do sistema que não desenvolveram, e podem ser usados os requisitos convencionais em vez de histórias de usuários. No entanto, a maioria das empresas que adotaram uma variante de XP usa *releases* de pequeno porte, desenvolvimento do *test-first* e integração contínua.

3.3.1 Teste em XP

Como discutido na introdução deste capítulo, uma das diferenças importantes entre o desenvolvimento incremental e o desenvolvimento dirigido a planos está na forma como o sistema é testado. Com o desenvolvimento incremental, não há especificação do sistema que possa ser usada por uma equipe de teste externa para desenvolvimento de testes do sistema. Como consequência, algumas abordagens para o desenvolvimento incremental têm um processo de testes muito informal em comparação com os testes dirigidos a planos.

Para evitar alguns dos problemas de teste e validação do sistema, a abordagem XP enfatiza a importância dos testes do programa. Extreme Programming inclui uma abordagem de testes que reduz as chances de erros desconhecidos na versão atual do sistema.

As principais características dos testes em XP são:

1. desenvolvimento *test-first*;
2. desenvolvimento de teste incremental a partir de cenários;
3. envolvimento dos usuários no desenvolvimento de testes e validação;
4. uso de *frameworks* de testes automatizados.

O desenvolvimento *test-first* é uma das mais importantes inovações no XP. Em vez de escrever algum código e, em seguida, escrever testes para esse código, você escreve os testes antes de escrever o código. Isso significa que você pode executar o teste enquanto o código está sendo escrito e pode encontrar problemas durante o desenvolvimento.

Ao escrever os testes, implicitamente se definem uma interface e uma especificação de comportamento para a funcionalidade a ser desenvolvida. Problemas de requisitos e mal-entendidos de interface são reduzidos. Essa abordagem pode ser adotada em qualquer processo em que haja uma relação clara entre um requisito do sistema e o código que implementa esse requisito. Em XP, você sempre pode ver esse link, porque os cartões de histórias que representam os requisitos são divididos em tarefas, e essas são a principal unidade de implementação. A adoção do desenvolvimento *test-first* em XP gerou o desenvolvimento de abordagens mais gerais dirigidas a testes (ASTELS, 2003). No Capítulo 8, discuto essas abordagens.

No desenvolvimento *test-first*, os implementadores de tarefas precisam entender completamente a especificação para que possam escrever testes para o sistema. Isso significa que as ambiguidades e omissões da lista de especificações devem ser esclarecidas antes do início da implementação. Além disso, também evita o problema de '*test-lag*'. Isso pode acontecer quando o desenvolvedor do sistema trabalha em um ritmo mais rápido que o

testador. A implementação fica mais e mais à frente dos testes e desenvolve-se uma tendência a ignorar os testes, a fim de que o cronograma de desenvolvimento possa ser mantido.

Em XP, os requisitos do usuário são expressos como cenários ou histórias, e o usuário os prioriza para o desenvolvimento. A equipe de desenvolvimento avalia cada cenário e divide-o em tarefas. Por exemplo, alguns dos cartões de tarefas desenvolvidos a partir do cartão de história para a prescrição de medicamentos (Quadro 3.1) são mostrados no Quadro 3.2. Cada tarefa gera um ou mais testes de unidade que verificam a implementação descrita naquela tarefa. O Quadro 3.3 é uma descrição resumida de um caso de teste desenvolvido para verificar se a dose prescrita de uma medicação não fica fora dos limites de segurança conhecidos.

No processo de testes, o papel do cliente é ajudar a desenvolver testes de aceitação para as histórias que serão implementadas no próximo *release* do sistema. Como discuto no Capítulo 8, o teste de aceitação é o processo em que o sistema é testado com os dados do cliente para verificar se o sistema atende às reais necessidades do cliente.

Em XP, o teste de aceitação, assim como o desenvolvimento, é incremental. O cliente, que faz parte da equipe, escreve os testes enquanto o desenvolvimento avança. Portanto, todos os novos códigos são validados para garantir que realmente é o que o cliente necessita. Para a história no Quadro 3.1, o teste de aceitação implicaria cenários nos quais (a) a dose de um medicamento foi alterada, (b) um novo medicamento foi selecionado e (c) o formulário foi usado para encontrar um medicamento. Na prática, em vez de um único teste, uma série de testes de aceitação é normalmente necessária.

Uma grande dificuldade no processo de teste em XP é contar com o apoio do cliente no desenvolvimento de testes de aceitação. Clientes têm muito pouco tempo disponível e podem não conseguir trabalhar com a equipe de desenvolvimento em tempo integral. O cliente pode sentir que fornecer os requisitos seja uma contribuição suficiente e, dessa forma, pode estar relutante em se envolver no processo de testes.

Automação de testes é essencial para o desenvolvimento *test-first*. Os testes são escritos como componentes executáveis antes que a tarefa seja implementada. Esses componentes de teste devem ser autônomos, devem simular a submissão de entrada a ser testada e devem verificar se o resultado atende à especificação de saída. Um *framework* de testes automatizados é um sistema que torna mais fácil escrever os testes executáveis e submeter um conjunto de testes para execução. Junit (MASSOL e HUSTED, 2003) é um exemplo amplamente usado de *framework* de testes automatizados.

Como o teste é automatizado, há sempre um conjunto de testes que podem ser executados rapidamente e com facilidade. Sempre que qualquer funcionalidade é adicionada ao sistema, os testes podem ser executados e os problemas que o novo código introduziu podem ser detectados imediatamente.

Desenvolvimento *test-first* e teste automatizado, geralmente, resultam em um grande número de testes sendo escritos e executados. No entanto, essa abordagem não leva, necessariamente, a testes completos do programa. Existem três razões para isso:

1. Programadores preferem programar para testes e, por vezes, tomam atalhos ao escrevê-los. Por exemplo, eles talvez escrevam testes incompletos que não verificam todas as possíveis exceções que podem ocorrer.
2. Alguns testes podem ser muito difíceis de escrever de forma incremental. Por exemplo, em uma interface complexa de usuário, muitas vezes é difícil escrever testes unitários para o código que implementa a 'lógica de exibição' e o *workflow* entre as telas.

Quadro 3.3 Descrição do caso de teste para verificação de dose

Teste 4: Verificação de dose
Entrada: 1. Um número em mg representando uma única dose da medicação. 2. Um número que representa o número de doses únicas por dia. **Testes:** 1. Teste para entradas em que a dose única é correta, mas a frequência é muito alta. 2. Teste para entradas em que a única dose é muito alta e muito baixa. 3. Teste para entradas em que a dose única x frequência é muito alta e muito baixa. 4. Teste para entradas em que a dose única x frequência é permitida. **Saída:** Mensagem de OK ou erro indicando que a dose está fora da faixa de segurança.

3. É difícil julgar a completude de um conjunto de testes. Embora você possa ter vários testes do sistema, o conjunto pode não fornecer uma cobertura completa. Partes essenciais do sistema podem não ser executadas e, assim, permanecer não testadas.

Portanto, apesar de um grande conjunto de testes executados frequentemente dar a impressão de que o sistema está completo e correto, esse pode não ser o caso. Se os testes não são revistos e outros testes não são escritos após o desenvolvimento, defeitos não detectados podem ser entregues no *release* do sistema.

3.3.2 A programação em pares

Outra prática inovadora introduzida no XP é que, para desenvolver o software, os programadores trabalhem em pares. Na verdade, para desenvolver o software eles se sentam juntos, na mesma estação de trabalho. No entanto, os mesmos pares nem sempre programam juntos. Pelo contrário, os pares são criados de maneira dinâmica, de modo que todos os membros da equipe trabalhem uns com os outros durante o processo de desenvolvimento.

O uso da programação em pares tem uma série de vantagens:

1. Dá suporte à ideia de propriedade e responsabilidade coletiva para o sistema, o que reflete a ideia de programação sem ego, de Weinberg (1971), segundo a qual o software é de propriedade da equipe como um todo e os indivíduos não são responsabilizados por problemas com o código. Em vez disso, a equipe tem responsabilidade coletiva para resolver esses problemas.
2. Atua como um processo de revisão informal, porque cada linha de código é observada por, pelo menos, duas pessoas. Inspeções e revisões do código (abordadas no Capítulo 24) são muito bem-sucedidas em descobrir uma elevada porcentagem de erros de softwares. No entanto, são demoradas para organizar e costumam apresentar atrasos no processo de desenvolvimento. Embora a programação em pares seja um processo menos formal que provavelmente não encontra tantos erros como as inspeções de código, é um processo de inspeção muito mais barato do que inspeções formais de programa.
3. Dá suporte à refatoração, que é um processo de melhoria de software. A dificuldade de implementar isso em um ambiente de desenvolvimento normal é que o esforço despendido na refatoração é para um benefício a longo prazo. Um indivíduo que pratica a refatoração pode ser considerado menos eficiente do que aquele que simplesmente atua no desenvolvimento de código. Sempre que a programação em pares e a propriedade coletiva são usadas, outros se beneficiam imediatamente da refatoração para que eles possam apoiar o processo.

Você pode pensar que a programação em pares é menos eficiente do que a individual. Em um período determinado, um par de desenvolvedores produziria metade da quantidade de código que dois indivíduos trabalhando sozinhos. Houve vários estudos a respeito da produtividade de programadores pagos, com resultados diversos. Usando estudantes voluntários, Williams e seus colaboradores (COCKBURN e WILLIAMS, 2001; WILLIAMS et al., 2000) concluíram que a produtividade na programação em pares parece ser comparável com a de duas pessoas que trabalham de forma independente. As razões sugeridas para tanto são as de que os pares discutem o software antes do desenvolvimento, de modo que provavelmente têm menos falsos começos e menos retrabalho. Além disso, o número de erros evitados pela inspeção informal reduz o tempo gasto consertando defeitos descobertos durante o processo de teste.

No entanto, estudos com programadores mais experientes (ARISHOLM et al., 2007; PARRISH et al., 2004) não replicaram esses resultados. Os pesquisadores descobriram uma perda significativa de produtividade em comparação com dois programadores trabalhando sozinhos. Havia alguns benefícios na qualidade, mas estes não compensaram o *overhead* da programação em pares. No entanto, o compartilhamento de conhecimento que acontece durante a programação em pares é muito importante, pois reduz os riscos globais para um projeto quando da saída de membros da equipe. Por si só, esse aspecto pode fazer a programação em pares valer a pena.

3.4 Gerenciamento ágil de projetos

A principal responsabilidade dos gerentes de projeto de software é gerenciar o projeto para que o software seja entregue no prazo e dentro do orçamento previsto. Eles supervisionam o trabalho dos engenheiros de software e acompanham quão bem o desenvolvimento de software está progredindo.

A abordagem-padrão para gerenciamento de projetos é a dirigida a planos. Como discuto no Capítulo 23, os gerentes devem elaborar um plano para o projeto mostrando o que deve ser entregue, quando deve ser en-

tregue e quem vai trabalhar no desenvolvimento das entregas do projeto. Uma abordagem baseada em planos necessita de um gerente que tenha uma visão estável de tudo o que tem de ser desenvolvido e os processos de desenvolvimento. Contudo, essa abordagem não funciona bem com os métodos ágeis, nos quais os requisitos são desenvolvidos de forma incremental, o software é entregue em incrementos curtos e rápidos, e as mudanças nos requisitos e no software são a norma.

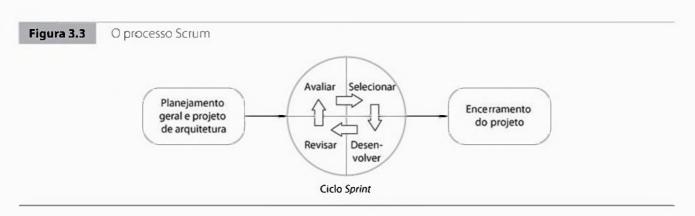
Como todos os outros processos profissionais de desenvolvimento de software, o desenvolvimento ágil tem de ser gerenciado de modo que se faça o melhor uso com o tempo e os recursos disponíveis para a equipe. Isso requer do gerenciamento de projeto uma abordagem diferente, adaptada para o desenvolvimento incremental e para os pontos fortes dos métodos ágeis.

A abordagem Scrum (SCHWABER, 2004; SCHWABER e BEEDLE, 2001) é um método ágil geral, mas seu foco está no gerenciamento do desenvolvimento iterativo, ao invés das abordagens técnicas específicas da engenharia de software ágil. A Figura 3.3 é um diagrama do processo Scrum de gerenciamento. Scrum não prescreve o uso de práticas de programação, como programação em pares e desenvolvimento *test-first*. Portanto, pode ser usado com abordagens ágeis mais técnicas, como XP, para fornecer um *framework* de gerenciamento do projeto.

No Scrum, existem três fases. A primeira é uma fase de planejamento geral, em que se estabelecem os objetivos gerais do projeto e da arquitetura do software. Em seguida, ocorre uma série de ciclos de *sprint*, sendo que cada ciclo desenvolve um incremento do sistema. Finalmente, a última fase do projeto encerra o projeto, completa a documentação exigida, como quadros de ajuda do sistema e manuais do usuário, e avalia as lições aprendidas com o projeto.

A característica inovadora do Scrum é sua fase central, chamada ciclos de *sprint*. Um *sprint* do Scrum é uma unidade de planejamento na qual o trabalho a ser feito é avaliado, os recursos para o desenvolvimento são selecionados e o software é implementado. No fim de um *sprint*, a funcionalidade completa é entregue aos *stakeholders*. As principais características desse processo são:

1. *Sprints* são de comprimento fixo, normalmente duas a quatro semanas. Eles correspondem ao desenvolvimento de um *release* do sistema em XP.
2. O ponto de partida para o planejamento é o *backlog* do produto, que é a lista do trabalho a ser feito no projeto. Durante a fase de avaliação do *sprint*, este é revisto, e as prioridades e os riscos são identificados. O cliente está intimamente envolvido nesse processo e, no início de cada *sprint*, pode introduzir novos requisitos ou tarefas.
3. A fase de seleção envolve todos da equipe do projeto que trabalham com o cliente para selecionar os recursos e a funcionalidade a ser desenvolvida durante o *sprint*.
4. Uma vez que todos estejam de acordo, a equipe se organiza para desenvolver o software. Reuniões diárias rápidas, envolvendo todos os membros da equipe, são realizadas para analisar os progressos e, se necessário, repriorizar o trabalho. Nessa etapa, a equipe está isolada do cliente e da organização, com todas as comunicações canalizadas por meio do chamado 'Scrum Master'. O papel do Scrum Master é proteger a equipe de desenvolvimento de distrações externas. A maneira como o trabalho é desenvolvido depende do problema e da equipe. Diferentemente do XP, a abordagem Scrum não faz sugestões específicas sobre como escrever os requisitos ou sobre o desenvolvimento *test-first* etc. No entanto, essas práticas de XP podem ser usadas se a equipe achar que são adequadas.
5. No fim do *sprint*, o trabalho é revisto e apresentado aos *stakeholders*. O próximo ciclo *sprint* começa em seguida.

Figura 3.3 O processo Scrum

A ideia por trás do Scrum é que toda a equipe deve ter poderes para tomar decisões, de modo que o termo 'gerente de projeto' tem sido deliberadamente evitado. Pelo contrário, o 'Scrum Master' é um facilitador, que organiza reuniões diárias, controla o *backlog* de trabalho, registra decisões, mede o progresso comparado ao *backlog* e se comunica com os clientes e a gerência externa à equipe.

Toda a equipe participa das reuniões diárias; às vezes, estas são feitas com os participantes em pé ('*stand-up*'), muito rápidas, para a manutenção do foco da equipe. Durante a reunião, todos os membros da equipe compartilham informações, descrevem seu progresso desde a última reunião, os problemas que têm surgido e o que está planejado para o dia seguinte. Isso garante que todos na equipe saibam o que está acontecendo e, se surgirem problemas, poderão replanejar o trabalho de curto prazo para lidar com eles. Todos participam desse planejamento de curto prazo; não existe uma hierarquia *top-down* a partir do Scrum Master.

Existem, na Internet, muitos relatos de sucesso do uso de Scrum. Rising e Janoff (2000) discutem seu uso bem-sucedido em um ambiente de desenvolvimento de software de telecomunicações, e listam suas vantagens conforme a seguir:

1. O produto é decomposto em um conjunto de partes gerenciáveis e compreensíveis.
2. Requisitos instáveis não atrasam o progresso.
3. Toda a equipe tem visão de tudo, e, consequentemente, a comunicação da equipe é melhorada.
4. Os clientes veem a entrega de incrementos dentro do prazo e recebem *feedback* sobre como o produto funciona.
5. Estabelece-se confiança entre clientes e desenvolvedores e cria-se uma cultura positiva, na qual todo mundo espera que o projeto tenha êxito.

O Scrum, como originalmente concebido, foi projetado para uso de equipes colocalizadas, em que todos os membros poderiam se encontrar todos os dias em reuniões rápidas. No entanto, muito do desenvolvimento de software atual envolve equipes distribuídas, ou seja, com membros da equipe situados em diferentes lugares ao redor do mundo. Consequentemente, estão em curso várias experiências para desenvolvimento Scrum para ambientes de desenvolvimento distribuído (SMITS e PSHIGODA, 2007; SUTHERLAND et al., 2007).

3.5 Escalamento de métodos ágeis

Os métodos ágeis foram desenvolvidos para serem usados por equipes de programação de pequeno porte que podiam trabalhar juntas na mesma sala e se comunicar de maneira informal. Os métodos ágeis foram, portanto, usados principalmente para o desenvolvimento de sistemas de pequeno e médio porte. Naturalmente, a necessidade de acelerar a entrega de software, adequada às necessidades do cliente, também se aplica a sistemas maiores. Consequentemente, tem havido um grande interesse em escalamento dos métodos ágeis para lidar com sistemas maiores, desenvolvidos por grandes organizações.

Denning e colegas. (2008) argumentam que a única maneira de evitar problemas comuns da engenharia de software, como os sistemas que não atendem às necessidades dos clientes e estouros de orçamento, é encontrar maneiras de fazer os métodos ágeis trabalharem para grandes sistemas. Leffingwell (2007) discute quais práticas de desenvolvimento ágil escalam para sistemas de grande porte. Moore e Spens (2008) relatam sua experiência em usar uma abordagem ágil para desenvolver um grande sistema médico com 300 desenvolvedores trabalhando em equipes distribuídas geograficamente.

O desenvolvimento de sistemas de software de grande porte é diferente do de sistemas pequenos, em vários pontos, como vemos a seguir.

1. Sistemas de grande porte geralmente são coleções de sistemas separados que se comunicam, nos quais equipes separadas desenvolvem cada um dos sistemas. Frequentemente, essas equipes estão trabalhando em lugares diferentes e, por vezes, em diferentes fusos horários. É praticamente impossível que cada equipe tenha uma visão de todo o sistema. Consequentemente, suas prioridades costumam ser voltadas para completar sua parte do sistema, sem levar em conta questões mais amplas do sistema como um todo.
2. Sistemas de grande porte são '*brownfield systems*' (HOPKINS e JENKINS, 2008), isto é, incluem e interagem com inúmeros sistemas existentes. Muitos dos requisitos do sistema estão preocupados com essa interação; assim, realmente não se prestam à flexibilidade e desenvolvimento incremental. Questões políticas também podem ser importantes aqui. Muitas vezes, a solução mais fácil para um problema é mudar um sistema em vigor. No

entanto, isso requer uma negociação com os gerentes do sistema para convencê-los de que as mudanças podem ser implementadas sem risco para a operação do sistema.

3. Sempre que vários sistemas estão integrados para criar um único, uma fração significativa do desenvolvimento preocupa-se com a configuração do sistema e não com o desenvolvimento do código original. Isso não é necessariamente compatível com o desenvolvimento incremental e com a integração frequente de sistemas.
4. Sistemas de grande porte e seus processos de desenvolvimento são frequentemente restringidos pelas regras externas e regulamentos que limitam o desenvolvimento, que exigem certos tipos de documentação a ser produzida etc.
5. Sistemas de grande porte têm um longo tempo de aquisição e desenvolvimento. É difícil manter equipes coerentes que saibam sobre o sistema durante esse período, pois as pessoas, inevitavelmente, deslocam-se para outros trabalhos e projetos.
6. Sistemas de grande porte geralmente têm um conjunto diverso de *stakeholders*. Por exemplo, enfermeiros e administradores podem ser os usuários finais de um sistema médico, mas o pessoal médico sênior, gerentes de hospital etc. também são *stakeholders* do sistema. É praticamente impossível envolver, no processo de desenvolvimento, todos esses diferentes *stakeholders*.

Há duas perspectivas no escalamento de métodos ágeis:

1. Perspectiva '*scaling up*', relacionada ao uso desses métodos para desenvolver sistemas de software de grande porte que não podem ser desenvolvidos por uma equipe pequena.
2. Perspectiva '*scaling out*', relacionada com a forma como os métodos ágeis podem ser introduzidos em uma grande organização com muitos anos de experiência em desenvolvimento de software.

Métodos ágeis precisam ser adaptados para lidar com sistemas de engenharia de grande porte. Leffingwell (2007) argumenta que é essencial manter os fundamentos dos métodos ágeis — planejamento flexível, frequentes *releases* do sistema, integração contínua, desenvolvimento dirigido a testes e boa comunicação entre os membros da equipe. Eu acredito que as adaptações críticas que necessitam ser introduzidas são as seguintes:

1. Para o desenvolvimento de sistemas de grande porte, não é possível focar apenas no código do sistema. Você precisa fazer mais projeto adiantado e documentação do sistema. A arquitetura de software precisa ser projetada, e é necessário haver documentos produzidos para descrever os aspectos críticos do sistema, como esquemas de banco de dados, a divisão de trabalho entre as equipes etc.
2. Mecanismos de comunicação entre equipes precisam ser projetados e usados. Isso deve envolver telefonemas e videoconferências regulares entre os membros da equipe, bem como reuniões eletrônicas frequentes e curtas nas quais os membros das diversas equipes se atualizam sobre o progresso uns dos outros. Uma série de canais de comunicação, como e-mail, mensagens instantâneas, *wikis* e sistemas de redes sociais, deve ser fornecida para facilitar as comunicações.
3. A integração contínua, em que todo o sistema é construído toda vez que um desenvolvedor verifica uma mudança, é praticamente impossível quando vários programas distintos precisam ser integrados para criar o sistema. No entanto, é essencial manter construções frequentes e *releases* regulares. Isso pode significar a necessidade da introdução de novas ferramentas de gerenciamento de configuração para suporte ao desenvolvimento de software com multiequipes.

Pequenas empresas que desenvolvem produtos de software estão entre os adeptos mais entusiastas dos métodos ágeis. Essas empresas não são limitadas pelas burocracias organizacionais ou padrões de processos e podem mudar rapidamente para adotar novas ideias. Naturalmente, as grandes empresas também têm feito experiências com métodos ágeis em projetos específicos, mas é muito mais difícil para eles introduzirem ('*scale out*') esses métodos em toda a organização. Lindvall e colegas. (2004) discutem alguns dos problemas em introduzir métodos ágeis em quatro grandes empresas de tecnologia.

A introdução de métodos ágeis em grandes empresas é difícil por diversas razões:

1. Os gerentes de projeto que não têm experiência em métodos ágeis podem ser relutantes em aceitar o risco de uma nova abordagem, uma vez que não sabem como isso vai afetar seus projetos particulares.
2. Nas grandes organizações existem procedimentos e padrões de qualidade que todos os projetos devem seguir e, por causa de sua natureza burocrática, geralmente são incompatíveis com os métodos ágeis. Às vezes, recebem suporte de ferramentas de software (por exemplo, ferramentas de gerenciamento de requisitos), e o uso dessas ferramentas é obrigatório a todos os projetos.

3. Métodos ágeis parecem funcionar melhor quando os membros da equipe têm um nível relativamente alto de habilidade. No entanto, dentro das grandes organizações é possível encontrar uma ampla gama de habilidades e competências; além disso, pessoas com níveis menores de habilidade podem não se tornar membros efetivos da equipe em processos ágeis.
4. Pode haver resistência cultural aos métodos ágeis, principalmente em organizações com longa história de uso dos processos convencionais de engenharia de sistemas.

Os procedimentos de gerenciamento de mudanças e testes são exemplos de procedimentos normais das empresas que podem não ser compatíveis com os métodos ágeis. O gerenciamento de mudanças é o processo de controle das mudanças em um sistema, de modo que o impacto das mudanças seja previsível, e os custos, controlados. Todas as mudanças necessitam de aprovação prévia antes de serem feitas, o que entra em conflito com a noção de refatoração. No XP, qualquer desenvolvedor pode melhorar qualquer código sem a aprovação externa. Para sistemas de grande porte, existem também padrões para os testes, em que uma construção de sistema é entregue a uma equipe externa de teste. Esse procedimento pode entrar em conflito com as abordagens *test-first* e testes frequentes, usadas em XP.

Apresentar e sustentar o uso de métodos ágeis em uma grande organização é um processo de mudança cultural. Mudanças culturais levam muito tempo para serem implementadas e, em muitos casos, demandam uma mudança de gerenciamento para serem efetivadas. As empresas que desejam usar métodos ágeis precisam de evangelizadores para promoverem as mudanças. Elas devem dedicar recursos significativos para o processo de mudança. Quando este livro foi escrito, poucas grandes empresas conseguiram fazer uma transição bem-sucedida para o desenvolvimento ágil em toda a organização.

PONTOS IMPORTANTES

- Métodos ágeis são métodos de desenvolvimento incremental que se concentram em desenvolvimento rápido, *releases* frequentes do software, redução de *overheads* dos processos e produção de códigos de alta qualidade. Eles envolvem o cliente diretamente no processo de desenvolvimento.
- A decisão de usar uma abordagem ágil ou uma abordagem dirigida a planos para o desenvolvimento deve depender do tipo de software a ser desenvolvido, das habilidades da equipe de desenvolvimento e da cultura da empresa que desenvolve o sistema.
- Extreme Programming é um método ágil, bem conhecido, que integra um conjunto de boas práticas de programação, como *releases* frequentes do software, melhorias contínuas do software e participação do cliente na equipe de desenvolvimento.
- Um ponto forte da Extreme Programming é o desenvolvimento de testes automatizados antes da criação de um recurso do programa. Quando um incremento é integrado ao sistema, todos os testes devem ser executados com sucesso.
- O método Scrum é uma metodologia ágil que fornece um *framework* de gerenciamento de projetos. É centralizado em torno de um conjunto de *sprints*, que são períodos determinados de tempo, quando um incremento de sistema é desenvolvido. O planejamento é baseado na priorização de um *backlog* de trabalho e na seleção das tarefas mais importantes para um *sprint*.
- O escalamento de métodos ágeis para sistemas de grande porte é difícil. Tais sistemas necessitam de projeto adiantado e alguma documentação. A integração contínua é praticamente impossível quando existem várias equipes de desenvolvimento separadas trabalhando em um projeto.

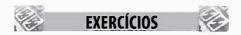
LEITURA COMPLEMENTAR

Extreme Programming Explained. Esse foi o primeiro livro sobre XP, e talvez ainda seja o mais lido. Ele explica a abordagem a partir da perspectiva de um de seus inventores, e seu entusiasmo fica explícito no livro. (BECK, K. *Extreme Programming Explained.* Addison-Wesley, 2000.)

Get Ready for Agile Methods, With Care. Uma crítica cuidadosa aos métodos ágeis, que discute seus pontos fortes e fracos, escrita por um engenheiro de software muito experiente. (BOEHM, B. *IEEE Computer*, jan. 2002.)

Scaling Software Agility: Best Practices for Large Enterprises. Embora com foco em questões de escalamento de desenvolvimento ágil, esse livro inclui também um resumo dos principais métodos ágeis, como XP, Scrum e Crystal. (LEFFINGWELL, D. *Scaling Software Agility: Best Practices for Large Enterprises.* Addison-Wesley, 2007.)

Running an Agile Software Development Project. A maioria dos livros sobre métodos ágeis se concentra em um método específico, mas esse livro tem uma abordagem diferente e discute como colocar XP em prática em um projeto. A obra apresenta conselhos bons e práticos. (HOLCOMBE, M. *Running an Agile Software Development Project.* John Wiley and Sons, 2008.)

EXERCÍCIOS

3.1 Explique por que, para as empresas, a entrega rápida e implantação de novos sistemas frequentemente é mais importante do que a funcionalidade detalhada desses sistemas.

3.2 Explique como os princípios básicos dos métodos ágeis levam ao desenvolvimento e implantação de software acelerados.

3.3 Quando você não recomendaria o uso de um método ágil para o desenvolvimento de um sistema de software?

3.4 Extreme Programming expressa os requisitos dos usuários como histórias, com cada história escrita em um cartão. Discuta as vantagens e desvantagens dessa abordagem para a descrição de requisitos.

3.5 Explique por que o desenvolvimento *test-first* ajuda o programador a desenvolver um melhor entendimento dos requisitos do sistema. Quais são as potenciais dificuldades com o desenvolvimento *test-first*?

3.6 Sugira quatro razões pelas quais a taxa de produtividade de programadores que trabalham em pares pode ser mais que a metade da taxa de produtividade de dois programadores que trabalham individualmente.

3.7 Compare e contraste a abordagem Scrum para o gerenciamento de projetos com abordagens convencionais dirigida a planos, como discutido no Capítulo 23. As comparações devem ser baseadas na eficácia de cada abordagem para o planejamento da alocação das pessoas nos projetos, estimativa de custos de projetos, manutenção da coesão da equipe e gerenciamento de mudanças no quadro da equipe do projeto.

3.8 Você é um gerente de software em uma empresa que desenvolve softwares críticos de controles para aeronaves. Você é responsável pelo desenvolvimento de um sistema de apoio ao projeto de software que dá suporte para a tradução de requisitos de software em uma especificação formal de software (discutido no Capítulo 13). Comente sobre as vantagens e desvantagens das estratégias de desenvolvimento a seguir:

a) Coletar dos engenheiros de software e *stakeholders* externos (como a autoridade regulatória de certificação) os requisitos para um sistema desse tipo e desenvolver o sistema usando uma abordagem dirigida a planos.

b) Desenvolver um protótipo usando uma linguagem de *script*, como Ruby ou Python, avaliar esse protótipo com os engenheiros de software e outros *stakeholders* e, em seguida, revisar os requisitos do sistema. Desenvolver novamente o sistema final, usando Java.

c) Desenvolver o sistema em Java, usando uma abordagem ágil com um usuário envolvido na equipe de desenvolvimento.

3.9 Tem-se sugerido que um dos problemas de se ter um usuário participando de uma equipe de desenvolvimento de software é que eles 'se tornam nativos', ou seja, adotam a perspectiva da equipe de desenvolvimento e perdem de vista as necessidades de seus colegas usuários. Sugira três maneiras de evitar esse problema e discuta as vantagens e desvantagens de cada abordagem.

3.10 Para reduzir os custos e o impacto ambiental das viagens, sua empresa decide fechar uma série de escritórios e dar suporte ao pessoal para trabalhar em casa. No entanto, a gerência sênior que introduz essa política não está ciente de que o software é desenvolvido por métodos ágeis, que contam com equipe trabalhando no mesmo local, e a programação em pares. Discuta as dificuldades que essa nova política pode causar e como você poderia contornar esses problemas.

REFERÊNCIAS

AMBLER, S. W.; JEFFRIES, R. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Nova York: John Wiley & Sons, 2002.

ARISHOLM, E.; GALLIS, H.; DYBA, T.; SJOBERG, D. I. K. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Trans*. on Software Eng., v. 33, n. 2, 2007, p. 65-86.

ASTELS, D. *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall, 2003.

BECK, K. Embracing Change with Extreme Programming. *IEEE Computer*, v. 32, n. 10, 1999, p. 70-78.

_______. *Extreme Programming Explained*. Reading, Mass.: Addison-Wesley, 2000.

CARLSON, D. *Eclipse Distilled*. Boston: Addison-Wesley, 2005.

COCKBURN, A. *Agile Software Development*. Reading, Mass.: Addison-Wesley, 2001.

_______. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Boston: Addison-Wesley, 2004.

COCKBURN, A.; WILLIAMS, L. The costs and benefits of pair programming. In: Extreme Programming Examined. Boston: Addison-Wesley, 2001.

COHN, M. *Succeeding with Agile: Software Development Using Scrum*. Boston: Addison-Wesley, 2009.

DEMARCO, T.; BOEHM, B. The Agile Methods Fray. *IEEE Computer*, v. 35, n. 6, 2002, p. 90-92.

DENNING, P. J.; GUNDERSON, C.; HAYES-ROTH, R. Evolutionary System Development. *Comm. ACM*, v. 51, n. 12, 2008, p. 29-31.

DROBNA, J.; NOFTZ, D.; RAGHU, R. Piloting XP on Four Mission-Critical Projects. *IEEE Software*, v. 21, n. 6, 2004, p. 70-75.

HIGHSMITH, J. A. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Nova York: Dorset House, 2000.

HOPKINS, R.; JENKINS, K. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston, Mass.: IBM Press, 2008.

LARMAN, C. *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Englewood Cliff, NJ: Prentice Hall, 2002.

LEFFINGWELL, D. *Scaling Software Agility: Best Practices for Large Enterprises*. Boston: Addison-Wesley, 2007.

LINDVALL, M.; MUTHIG, D.; DAGNINO, A.; WALLIN, C.; STUPPERICH, M.; KIEFER, D. et al. Agile Software Development in Large Organizations. *IEEE Computer*, v. 37, n. 12, 2004, p. 26-34.

MARTIN, J. *Application Development Without Programmers*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

MASSOL, V.; HUSTED, T. *JUnit in Action*. Greenwich, Conn.: Manning Publications Co., 2003.

MILLS, H. D.; O'NEILL, D.; LINGER, R. C.; DYER, M.; QUINNAN, R. E. The Management of Software Engineering. *IBM Systems. J.*, .v 19, n. 4, 1980, p. 414-477.

MOORE, E.; SPENS, J. Scaling Agile: Finding your Agile Tribe. Proc. Agile 2008 Conference. *IEEE Computer Society*, Toronto, 2008, p. 121-124.

PALMER, S. R.; FELSING, J. M. *A Practical Guide to Feature-Driven Development*. Englewood Cliffs, NJ: Prentice Hall, 2002.

PARRISH, A.; SMITH, R.; HALE, D.; HALE, J. A Field Study of Developer Pairs: Productivity Impacts and Implications. *IEEE Software*, v. 21, n. 5, 2004, p. 76-79.

POOLE, C.; HUISMAN, J. W. Using Extreme Programming in a Maintenance Environment. *IEEE Software*, v. 18, n. 6, 2001, p. 42-50.

RISING, L.; JANOFF, N. S. The Scrum Software Development Process for Small Teams. *IEEE Software*, v. 17, n. 4, 2000, p. 26-32.

SCHWABER, K. *Agile Project Management with Scrum*. Seattle: Microsoft Press, 2004.

SCHWABER, K.; BEEDLE, M. *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice Hall, 2001.

SMITS, H.; PSHIGODA, G. Implementing Scrum in a Distributed Software Development Organization. *Agile 2007*. Washington, DC: IEEE Computer Society, 2007.

STAPLETON, J. *DSDM Dynamic Systems Development Method*. Harlow, Reino Unido: Addison-Wesley, 1997.

_______. *DSDM: Business Focused Development*, 2. ed. Harlow, Reino Unido: Pearson Education, 2003.

STEPHENS, M.; ROSENBERG, D. *Extreme Programming Refactored*. Berkley, Califórnia: Apress, 2003.

SUTHERLAND, J.; VIKTOROV, A.; BLOUNT, J.; PUNTIKOV, N. Distributed Scrum: Agile Project Management with Outsourced Development Teams. 40th Hawaii Int. Conf. on System Sciences. *IEEE Computer Society*, Hawaii, 2007.

WEINBERG, G. *The Psychology of Computer Programming*. Nova York: Van Nostrand, 1971.

WILLIAMS, L.; KESSLER, R. R.; CUNNINGHAM, W.; JEFFRIES, R. Strengthening the Case for Pair Programming. *IEEE Software*, v. 17, n. 4, 2000, p. 19-25.

CAPÍTULO

4

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Engenharia de requisitos

Objetivos

O objetivo deste capítulo é apresentar os requisitos de software e discutir os processos envolvidos na descoberta e documentação desses requisitos. Após a leitura, você:

- compreenderá os conceitos de requisitos de usuário e de sistema e por que eles devem ser escritos de formas diferentes;
- compreenderá as diferenças entre requisitos de software funcionais e não funcionais;
- compreenderá como os requisitos podem ser organizados em um documento de requisitos de software;
- compreenderá as principais atividades de elicitação, análise e validação da engenharia de requisitos e as relações entre essas atividades;
- compreenderá por que o gerenciamento de requisitos é necessário e como ele dá suporte às outras atividades da engenharia de requisitos.

Conteúdo

Os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento. Esses requisitos refletem as necessidades dos clientes para um sistema que serve a uma finalidade determinada, como controlar um dispositivo, colocar um pedido ou encontrar informações. O processo de descobrir, analisar, documentar e verificar esses serviços e restrições é chamado engenharia de requisitos (RE, do inglês *requirements engineering*).

O termo 'requisito' não é usado de forma consistente pela indústria de software. Em alguns casos, o requisito é apenas uma declaração abstrata em alto nível de um serviço que o sistema deve oferecer ou uma restrição a um sistema. No outro extremo, é uma definição detalhada e formal de uma função do sistema. Davis (1993) explica por que essas diferenças existem:

> *Se uma empresa pretende fechar um contrato para um projeto de desenvolvimento de software de grande porte, deve definir as necessidades de forma abstrata o suficiente para que a solução para essas necessidades não seja predefinida. Os requisitos precisam ser escritos de modo que vários contratantes possam concorrer pelo contrato e oferecer diferentes maneiras de atender às necessidades da organização do cliente. Uma vez que o contrato tenha sido adjudicado, o contratante deve escrever para o cliente uma definição mais detalhada do sistema, para que este entenda e possa validar o que o software fará. Ambos os documentos podem ser chamados documentos de requisitos para o sistema.*

Alguns dos problemas que surgem durante o processo de engenharia de requisitos são as falhas em não fazer uma clara separação entre esses diferentes níveis de descrição. Faço uma distinção entre eles usando o termo 'requisitos de

usuário', para expressar os requisitos abstratos de alto nível, e 'requisitos de sistema', para expressar a descrição detalhada do que o sistema deve fazer. Requisitos de usuário e requisitos de sistema podem ser definidos como segue:

1. Requisitos de usuário são declarações, em uma linguagem natural com diagramas, de quais serviços o sistema deverá fornecer a seus usuários e as restrições com as quais este deve operar.
2. Requisitos de sistema são descrições mais detalhadas das funções, serviços e restrições operacionais do sistema de software. O documento de requisitos do sistema (às vezes, chamado especificação funcional) deve definir exatamente o que deve ser implementado. Pode ser parte do contrato entre o comprador do sistema e os desenvolvedores de software.

Diferentes níveis de requisitos são úteis, pois eles comunicam informações sobre o sistema para diferentes tipos de leitor. A Figura 4.1 ilustra a distinção entre requisitos de usuário e de sistema. Esse exemplo, de um Sistema de Gerenciamento da Saúde Mental de Pacientes (MHC-PMS, do inglês *Mental Health Care Patient Management System*), mostra como um requisito de usuário pode ser expandido em diversos requisitos de sistemas. Pode-se ver na Figura 4.1 que os requisitos de usuário são mais gerais. Os requisitos de sistema fornecem informações mais específicas sobre os serviços e funções do sistema que devem ser implementados.

Os requisitos precisam ser escritos em diferentes níveis de detalhamento para que diferentes leitores possam usá-los de diversas maneiras. A Figura 4.2 mostra possíveis leitores dos requisitos de usuário e de sistema. Os leitores dos requisitos de usuário não costumam se preocupar com a forma como o sistema será implementado; podem ser gerentes que não estão interessados nos recursos detalhados do sistema. Os leitores dos requisitos de sistema precisam saber mais detalhadamente o que o sistema fará, porque estão interessados em como ele apoiará os processos dos negócios ou porque estão envolvidos na implementação do sistema.

Neste capítulo, apresento uma visão 'tradicional' de requisitos e não de requisitos em processos ágeis. Para a maioria dos sistemas de grande porte, ainda é o caso de existir uma fase da engenharia de requisitos claramente identificável antes de se iniciar a implementação do sistema. O resultado é um documento de requisitos, que pode ser parte do contrato de desenvolvimento do sistema. Certamente, haverá mudanças nos requisitos e os requisitos de usuário poderão ser ampliados em requisitos de sistema mais detalhados. No entanto, a abordagem ágil de simultaneamente elicitar os requisitos enquanto o sistema é desenvolvido é raramente utilizada para desenvolvimento de sistemas de grande porte.

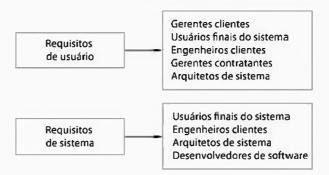
Figura 4.1 Requisitos de usuário e de sistema

Definição de requisitos de usuário

1. O MHC-PMS deve gerar relatórios gerenciais mensais que mostrem o custo dos medicamentos prescritos por cada clínica durante aquele mês.

Especificação de requisitos de sistema

1.1 No último dia útil de cada mês deve ser gerado um resumo dos medicamentos prescritos, seus custos e as prescrições de cada clínica.
1.2 Após 17:30h do último dia útil do mês, o sistema deve gerar automaticamente o relatório para impressão.
1.3 Um relatório será criado para cada clínica, listando os nomes dos medicamentos, o número total de prescrições, o número de doses prescritas e o custo total dos medicamentos prescritos.
1.4 Se os medicamentos estão disponíveis em diferentes unidades de dosagem (por exemplo, 10 mg, 20 mg), devem ser criados relatórios separados para cada unidade.
1.5 O acesso aos relatórios de custos deve ser restrito a usuários autorizados por uma lista de controle de gerenciamento de acesso.

Figura 4.2 Leitores de diferentes tipos de especificação de requisitos

4.1 Requisitos funcionais e não funcionais

Os requisitos de software são frequentemente classificados como requisitos funcionais e requisitos não funcionais:

1. *Requisitos funcionais.* São declarações de serviços que o sistema deve fornecer, de como o sistema deve reagir a entradas específicas e de como o sistema deve se comportar em determinadas situações. Em alguns casos, os requisitos funcionais também podem explicitar o que o sistema não deve fazer.
2. *Requisitos não funcionais.* São restrições aos serviços ou funções oferecidos pelo sistema. Incluem restrições de *timing*, restrições no processo de desenvolvimento e restrições impostas pelas normas. Ao contrário das características individuais ou serviços do sistema, os requisitos não funcionais, muitas vezes, aplicam-se ao sistema como um todo.

Na realidade, a distinção entre diferentes tipos de requisitos não é tão clara como sugerem essas definições simples. Um requisito de usuário relacionado com a proteção, tal como uma declaração de limitação de acesso a usuários autorizados, pode parecer um requisito não funcional. No entanto, quando desenvolvido em mais detalhes, esse requisito pode gerar outros requisitos, claramente funcionais, como a necessidade de incluir recursos de autenticação de usuário no sistema.

Isso mostra que os requisitos não são independentes e que muitas vezes geram ou restringem outros requisitos. Portanto, os requisitos de sistema não apenas especificam os serviços ou as características necessárias ao sistema, mas também a funcionalidade necessária para garantir que esses serviços/características sejam entregues corretamente.

4.1.1 Requisitos funcionais

Os requisitos funcionais de um sistema descrevem o que ele deve fazer. Eles dependem do tipo de software a ser desenvolvido, de quem são seus possíveis usuários e da abordagem geral adotada pela organização ao escrever os requisitos. Quando expressos como requisitos de usuário, os requisitos funcionais são normalmente descritos de forma abstrata, para serem compreendidos pelos usuários do sistema. No entanto, requisitos de sistema funcionais mais específicos descrevem em detalhes as funções do sistema, suas entradas e saídas, exceções etc.

Requisitos funcionais do sistema variam de requisitos gerais, que abrangem o que o sistema deve fazer, até requisitos muito específicos, que refletem os sistemas e as formas de trabalho em uma organização. Por exemplo, aqui estão os exemplos de requisitos funcionais para o sistema MHC-PMS, usados para manter informações sobre os pacientes em tratamento por problemas de saúde mental:

1. Um usuário deve ser capaz de pesquisar as listas de agendamentos para todas as clínicas.
2. O sistema deve gerar a cada dia, para cada clínica, a lista dos pacientes para as consultas daquele dia.
3. Cada membro da equipe que usa o sistema deve ser identificado apenas por seu número de oito dígitos.

Esses requisitos funcionais dos usuários definem os recursos específicos a serem fornecidos pelo sistema. Eles foram retirados do documento de requisitos de usuário e mostram que os requisitos funcionais podem ser escritos em diferentes níveis de detalhamento (contrastar requisitos 1 e 3).

A imprecisão na especificação de requisitos é a causa de muitos problemas da engenharia de software. É compreensível que um desenvolvedor de sistemas interprete um requisito ambíguo de uma maneira que simplifique sua implementação. Muitas vezes, porém, essa não é a preferência do cliente, sendo necessário, então, estabelecer novos requisitos e fazer alterações no sistema. Naturalmente, esse procedimento gera atrasos de entrega e aumenta os custos.

Por exemplo, o primeiro requisito hipotético para o MHC-PMS diz que um usuário deve ser capaz de buscar as listas de agendamentos para todas as clínicas. A justificativa para esse requisito é que os pacientes com problemas de saúde mental por vezes são confusos. Eles podem ter uma consulta em uma clínica, mas se deslocarem até outra. Caso eles tenham uma consulta marcada, eles serão registrados como atendidos, independente da clínica.

O membro da equipe médica que especifica esse requisito pode esperar que 'pesquisar' signifique que, dado um nome de paciente, o sistema vai procurar esse nome em todos os agendamentos de todas as clínicas. No entanto, isso não está explícito no requisito. Desenvolvedores do sistema podem interpretar o requisito de maneira diferente e implementar uma pesquisa em que o usuário tenha de escolher uma clínica, e, em seguida, realizar a pesquisa. Isso, obviamente, envolverá mais entradas do usuário e necessitará de mais tempo.

Em princípio, a especificação dos requisitos funcionais de um sistema deve ser completa e consistente. Completude significa que todos os serviços requeridos pelo usuário devem ser definidos. Consistência significa que os requisitos não devem ter definições contraditórias. Na prática, para sistemas grandes e complexos, é praticamente impossível alcançar completude e consistência dos requisitos. Uma razão para isso é que ao elaborar especificações para sistemas complexos é fácil cometer erros e omissões. Outra razão é que em um sistema de grande porte existem muitos *stakeholders*. Um *stakeholder* é uma pessoa ou papel que, de alguma maneira, é afetado pelo sistema. Os *stakeholders* têm necessidades diferentes — e, muitas vezes, inconsistentes. Essas inconsistências podem não ser evidentes em um primeiro momento, quando os requisitos são especificados, e, assim, são incluídas na especificação. Os problemas podem surgir após uma análise mais profunda ou depois de o sistema ter sido entregue ao cliente.

4.1.2 Requisitos não funcionais

Os requisitos não funcionais, como o nome sugere, são requisitos que não estão diretamente relacionados com os serviços específicos oferecidos pelo sistema a seus usuários. Eles podem estar relacionados às propriedades emergentes do sistema, como confiabilidade, tempo de resposta e ocupação de área. Uma alternativa a esse cenário seria os requisitos definirem restrições sobre a implementação do sistema, como as capacidades dos dispositivos de E/S ou as representações de dados usadas nas interfaces com outros sistemas.

Os requisitos não funcionais, como desempenho, proteção ou disponibilidade, normalmente especificam ou restringem as características do sistema como um todo. Requisitos não funcionais são frequentemente mais críticos que requisitos funcionais individuais. Os usuários do sistema podem, geralmente, encontrar maneiras de contornar uma função do sistema que realmente não atenda a suas necessidades. No entanto, deixar de atender a um requisito não funcional pode significar a inutilização de todo o sistema. Por exemplo, se um sistema de aeronaves não cumprir seus requisitos de confiabilidade, não será certificado como um sistema seguro para operar; se um sistema de controle embutido não atender aos requisitos de desempenho, as funções de controle não funcionarão corretamente.

Embora muitas vezes seja possível identificar quais componentes do sistema implementam requisitos funcionais específicos (por exemplo, pode haver componentes de formatação que implementam requisitos de impressão de relatórios), é frequentemente mais difícil relacionar os componentes com os requisitos não funcionais. A implementação desses requisitos pode ser difundida em todo o sistema. Há duas razões para isso:

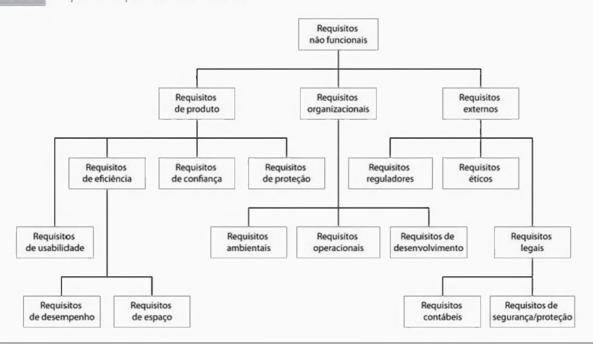
1. Requisitos não funcionais podem afetar a arquitetura geral de um sistema em vez de apenas componentes individuais. Por exemplo, para assegurar que sejam cumpridos os requisitos de desempenho, será necessário organizar o sistema para minimizar a comunicação entre os componentes.
2. Um único requisito não funcional, tal como um requisito de proteção, pode gerar uma série de requisitos funcionais relacionados que definam os serviços necessários no novo sistema. Além disso, também podem gerar requisitos que restrinjam requisitos existentes.

Os requisitos não funcionais surgem por meio das necessidades dos usuários, devido a restrições de orçamento, políticas organizacionais, necessidade de interoperabilidade com outros sistemas de software ou hardware, ou a partir de fatores externos, como regulamentos de segurança ou legislações de privacidade. A Figura 4.3 é uma classificação de requisitos não funcionais. Nesse diagrama você pode ver que os requisitos não funcionais podem ser provenientes das características requeridas para o software (requisitos de produto), da organização que desenvolve o software (requisitos organizacionais) ou de fontes externas:

1. *Requisitos de produto.* Esses requisitos especificam ou restringem o comportamento do software. Exemplos incluem os requisitos de desempenho quanto à rapidez com que o sistema deve executar e quanta memória ele requer, os requisitos de confiabilidade que estabelecem a taxa aceitável de falhas, os requisitos de proteção e os requisitos de usabilidade.
2. *Requisitos organizacionais.* Esses são os requisitos gerais de sistemas derivados das políticas e procedimentos da organização do cliente e do desenvolvedor. Exemplos incluem os requisitos do processo operacional, que definem como o sistema será usado, os requisitos do processo de desenvolvimento que especificam a linguagem de programação, o ambiente de desenvolvimento ou normas de processo a serem usadas, bem como os requisitos ambientais que especificam o ambiente operacional do sistema.
3. *Requisitos externos.* Esse tipo abrange todos os requisitos que derivam de fatores externos ao sistema e seu processo de desenvolvimento. Podem incluir requisitos reguladores, que definem o que deve ser feito para que o sistema seja aprovado para uso, por um regulador, tal como um banco central; requisitos legais, que devem ser seguidos para garantir que o sistema opere dentro da lei; e requisitos éticos, que asseguram que o sistema será aceitável para seus usuários e o público em geral.

O Quadro 4.1 mostra exemplos de requisitos de produto, organizacional e externo retirados do MHC-PMS, cujos requisitos de usuário foram introduzidos na Seção 4.1.1. O requisito de produto é um requisito de disponibilidade que define quando o sistema deve estar disponível e o tempo diário permitido de seu não funcionamento. Não trata da funcionalidade do MHC-PMS e identifica claramente uma restrição que deve ser considerada pelos projetistas do sistema.

O requisito organizacional especifica como os usuários se autenticam para o sistema. A autoridade de saúde que opera o sistema está migrando para um procedimento de autenticação-padrão para todos os softwares. Neste, em vez de o usuário ter um nome de *login* para se identificar, ele deve passar seu cartão de identificação por um leitor. O requisito externo deriva da necessidade de o sistema estar em conformidade com a legislação de privacidade. A privacidade é obviamente uma questão muito importante nos sistemas de saúde e o requisito especifica claramente que o sistema deverá ser desenvolvido de acordo com uma norma nacional de privacidade.

Figura 4.3 Tipos de requisitos não funcionais

Quadro 4.1 Exemplos de requisitos não funcionais no MHC-PMS.

Requisito de produto
O MHC-PMS deve estar disponível para todas as clínicas durante as horas normais de trabalho (segunda a sexta-feira, 8h30 às 17h30). Períodos de não operação dentro do horário normal de trabalho não podem exceder cinco segundos em um dia.

Requisito organizacional
Usuários do sistema MHC-PMS devem se autenticar com seus cartões de identificação da autoridade da saúde.

Requisito externo
O sistema deve implementar as disposições de privacidade dos pacientes, tal como estabelecido no HStan-03-2006-priv.

Um problema comum com os requisitos não funcionais é que costumam ser propostos pelos usuários ou clientes como metas gerais, em virtude da facilidade de uso, da capacidade do sistema de se recuperar de falhas ou da velocidade das respostas do usuário. Metas estabelecem boas intenções, mas podem causar problemas para os desenvolvedores do sistema, uma vez que deixam margem para interpretação e, consequentemente, para disputas, quando da entrega do sistema. A meta do sistema apresentado a seguir, por exemplo, é típica de como um gerente pode expressar requisitos de usabilidade:

O sistema deve ser de fácil uso pelo pessoal médico e deve ser organizado de tal maneira que os erros dos usuários sejam minimizados.

Reescrevi apenas para mostrar como essa meta poderia ser expressa como um requisito não funcional 'testável'. É impossível verificar objetivamente a finalidade do sistema, mas na descrição a seguir é possível incluir pelo menos a instrumentação de software para contar os erros cometidos pelos usuários quando estão testando o sistema.

A equipe médica deve ser capaz de usar todas as funções do sistema após quatro horas de treinamento. Após esse treinamento, o número médio de erros cometidos por usuários experientes não deve exceder dois por hora de uso do sistema.

Sempre que possível, os requisitos não funcionais devem ser escritos quantitativamente, para que possam ser objetivamente testados. A Tabela 4.1 mostra as métricas que você pode usar para especificar as propriedades não funcionais do sistema. Você pode medir essas características quando o sistema está sendo testado para verificar se ele tem cumprido ou não seus requisitos não funcionais.

Na prática, os clientes de um sistema geralmente consideram difícil traduzir suas metas em requisitos mensuráveis. Para algumas metas, como manutenibilidade, não existem métricas que possam ser usadas. Em outros casos, mesmo quando a especificação quantitativa é possível, os clientes podem não ser capazes de relacionar suas necessidades com essas especificações. Eles não entendem o que significa um número definindo a confiabilidade necessária (por exemplo), em termos de sua experiência cotidiana com os sistemas computacionais. Além disso, o custo de verificar requisitos objetivamente não funcionais mensuráveis pode ser muito elevado, e os clientes, que pagam pelo sistema, podem não achar que os custos sejam justificados.

Os requisitos não funcionais frequentemente conflitam e interagem com outros requisitos funcionais ou não funcionais. Por exemplo, o requisito de autenticação no Quadro 4.1 certamente exige a instalação de um leitor de cartão em cada computador conectado ao sistema. No entanto, pode haver outro requisito que solicite acesso móvel ao sistema, pelos laptops dos médicos ou enfermeiras. Estes não são geralmente equipados com leitores de cartão, assim, nessas circunstâncias, algum método de autenticação alternativo deve ser criado.

Na prática, no documento de requisitos, é difícil separar os requisitos funcionais dos não funcionais. Se são apresentados separadamente, os relacionamentos entre eles podem ficar difíceis de serem entendidos. No entanto, os requisitos claramente relacionados com as propriedades emergentes do sistema, como desempenho ou confiabilidade, devem ser explicitamente destacados. Você pode fazer isso colocando-os em uma seção separada do documento de requisitos ou distinguindo-os, de alguma forma, dos outros requisitos de sistema.

Requisitos não funcionais, como confiabilidade, segurança e confidencialidade, são particularmente importantes para sistemas críticos. Escrevo sobre esses requisitos no Capítulo 12, no qual descrevo técnicas próprias para a especificação de requisitos de confiança e de proteção.

Tabela 4.1 Métricas para especificar requisitos não funcionais.

Propriedade	Medida
Velocidade	Transações processadas/segundo Tempo de resposta de usuário/evento Tempo de atualização de tela
Tamanho	Megabytes Número de chips de memória ROM
Facilidade de uso	Tempo de treinamento Número de *frames* de ajuda
Confiabilidade	Tempo médio para falha Probabilidade de indisponibilidade Taxa de ocorrência de falhas Disponibilidade
Robustez	Tempo de reinício após falha Percentual de eventos que causam falhas Probabilidade de corrupção de dados em caso de falha
Portabilidade	Percentual de declarações dependentes do sistema-alvo Número de sistemas-alvo

4.2 O documento de requisitos de software

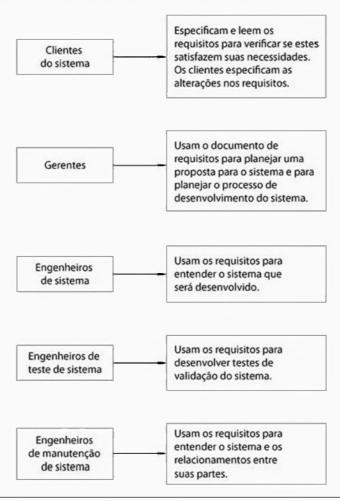
O documento de requisitos de software, às vezes chamado Especificação de Requisitos de Software (SRS — do inglês *Software Requirements Specification*), é uma declaração oficial de o que os desenvolvedores do sistema devem implementar. Deve incluir tanto os requisitos de usuário para um sistema quanto uma especificação detalhada dos requisitos de sistema. Em alguns casos, os requisitos de usuário e de sistema são integrados em uma única descrição. Em outros, os requisitos de usuário são definidos em uma introdução à especificação de requisitos de sistema. Se houver um grande número de requisitos, os requisitos detalhados de sistema podem ser apresentados em um documento separado.

Documentos de requisitos são essenciais quando um contratante externo está desenvolvendo o sistema de software. Entretanto, os métodos ágeis de desenvolvimento argumentam que os requisitos mudam tão rapidamente que um documento de requisitos já está ultrapassado assim que termina de ser escrito. Portanto, o esforço é, em grande parte, desperdiçado. Em vez de um documento formal, abordagens como a Extreme Programming (BECK, 1999) coletam os requisitos de usuário de forma incremental e escrevem-nos em cartões como estórias de usuário. O usuário então prioriza os requisitos para implementação no próximo incremento do sistema.

Acredito que essa seja uma boa abordagem para os sistemas de negócio em que os requisitos são instáveis. No entanto, penso que ainda é útil escrever um pequeno documento de apoio no qual estejam definidos os requisitos de negócio e de confiança para o sistema; quando o foco está nos requisitos funcionais dos próximos *releases* do sistema, é fácil nos esquecermos dos requisitos que se aplicam ao sistema como um todo.

O documento de requisitos tem um conjunto diversificado de usuários, que vão desde a alta administração da organização que está pagando pelo sistema até os engenheiros responsáveis pelo desenvolvimento do software. A Figura 4.4, tirada do meu livro com Gerald Kotonya sobre engenharia de requisitos (KOTONYA e SOMMERVILLE, 1998) mostra os possíveis usuários do documento e como eles o utilizam.

A diversidade de possíveis usuários é um indicativo de que o documento de requisitos precisa ser um compromisso com a comunicação dos requisitos para os clientes, a definição dos requisitos em detalhes precisos para os desenvolvedores e testadores e a inclusão de informações sobre a possível evolução do sistema. Informações sobre mudanças previstas podem ajudar os projetistas de sistema a evitar decisões de projeto restritivas, além de ajudar os engenheiros de manutenção de sistema que precisam adaptar o sistema aos novos requisitos.

Figura 4.4 Usuários de um documento de engenharia de requisitos.

O nível de detalhes que você deve incluir em um documento de requisitos depende do tipo de sistema em desenvolvimento e o processo usado. Os sistemas críticos precisam ter requisitos detalhados, porque a segurança e a proteção devem ser analisadas em detalhes. Quando o sistema está sendo desenvolvido por uma companhia separada (por exemplo, através de *outsourcing*), as especificações de sistema devem ser detalhadas e precisas. Se um processo interno de desenvolvimento iterativo é usado, o documento de requisitos pode ser muito menos detalhado e quaisquer ambiguidades podem ser resolvidas durante o desenvolvimento do sistema.

A Tabela 4.2 mostra uma possível organização de um documento de requisitos baseada em uma norma IEEE para documentos de requisitos (IEEE, 1998). Essa é uma norma genérica que pode ser adaptada para usos específicos. Nesse caso, eu estendi a norma para incluir informações sobre a evolução prevista do sistema. Essa informação ajuda os engenheiros de manutenção de sistema e permite que os projetistas incluam suporte para futuros recursos do sistema.

Naturalmente, a informação incluída em um documento de requisitos depende do tipo de software a ser desenvolvido e da abordagem de desenvolvimento que está em uso. Se uma abordagem evolutiva é adotada para um produto de software (por exemplo), o documento de requisitos deixará de fora muitos dos capítulos detalhados sugeridos. O foco será sobre a definição de requisitos de usuário e os requisitos não funcionais de alto nível de sistema. Nesse caso, os projetistas e programadores usam seu julgamento para decidir como atender aos requisitos gerais de usuário para o sistema.

No entanto, quando o software é parte de um projeto de um sistema de grande porte que inclui interações entre sistemas de hardware e software, geralmente é necessário definir os requisitos em um alto nível de detalhamento. Isso significa que esses documentos de requisitos podem ser muito longos e devem incluir a maioria ou todos os capítulos mostrados na Tabela 4.2. É particularmente importante incluir uma tabela completa, abrangendo conteúdo e índice de documentos, para que os leitores possam encontrar as informações de que necessitam.

Tabela 4.2 A estrutura de um documento de requisitos.

Capítulo	Descrição
Prefácio	Deve definir os possíveis leitores do documento e descrever seu histórico de versões, incluindo uma justificativa para a criação de uma nova versão e um resumo das mudanças feitas em cada versão.
Introdução	Deve descrever a necessidade para o sistema. Deve descrever brevemente as funções do sistema e explicar como ele vai funcionar com outros sistemas. Também deve descrever como o sistema atende aos objetivos globais de negócio ou estratégicos da organização que encomendou o software.
Glossário	Deve definir os termos técnicos usados no documento. Você não deve fazer suposições sobre a experiência ou o conhecimento do leitor.
Definição de requisitos de usuário	Deve descrever os serviços fornecidos ao usuário. Os requisitos não funcionais de sistema também devem ser descritos nessa seção. Essa descrição pode usar a linguagem natural, diagramas ou outras notações compreensíveis para os clientes. Normas de produto e processos que devem ser seguidos devem ser especificados.
Arquitetura do sistema	Deve apresentar uma visão geral em alto nível da arquitetura do sistema previsto, mostrando a distribuição de funções entre os módulos do sistema. Componentes de arquitetura que são reusados devem ser destacados.
Especificação de requisitos do sistema	Deve descrever em detalhes os requisitos funcionais e não funcionais. Se necessário, também podem ser adicionados mais detalhes aos requisitos não funcionais. Interfaces com outros sistemas podem ser definidas.
Modelos do sistema	Pode incluir modelos gráficos do sistema que mostram os relacionamentos entre os componentes do sistema, o sistema e seu ambiente. Exemplos de possíveis modelos são modelos de objetos, modelos de fluxo de dados ou modelos semânticos de dados.
Evolução do sistema	Deve descrever os pressupostos fundamentais em que o sistema se baseia, bem como quaisquer mudanças previstas, em decorrência da evolução de hardware, de mudanças nas necessidades do usuário etc. Essa seção é útil para projetistas de sistema, pois pode ajudá-los a evitar decisões capazes de restringir possíveis mudanças futuras no sistema.
Apêndices	Deve fornecer informações detalhadas e específicas relacionadas à aplicação em desenvolvimento, além de descrições de hardware e banco de dados, por exemplo. Os requisitos de hardware definem as configurações mínimas ideais para o sistema. Requisitos de banco de dados definem a organização lógica dos dados usados pelo sistema e os relacionamentos entre esses dados.
Índice	Vários índices podem ser incluídos no documento. Pode haver, além de um índice alfabético normal, um índice de diagramas, de funções, entre outros pertinentes.

4.3 Especificação de requisitos

A especificação de requisitos é o processo de escrever os requisitos de usuário e de sistema em um documento de requisitos. Idealmente, os requisitos de usuário e de sistema devem ser claros, inequívocos, de fácil compreensão, completos e consistentes. Na prática, isso é difícil de conseguir, pois os *stakeholders* interpretam os requisitos de maneiras diferentes, e, muitas vezes, notam-se conflitos e inconsistências inerentes aos requisitos.

Os requisitos de usuário para um sistema devem descrever os requisitos funcionais e não funcionais de modo que sejam compreensíveis para os usuários do sistema que não tenham conhecimentos técnicos detalhados. Idealmente, eles devem especificar somente o comportamento externo do sistema. O documento de requisitos não deve incluir detalhes da arquitetura ou projeto do sistema. Consequentemente, se você está escrevendo requisitos de usuário, não deve usar o jargão de software, notações estruturadas ou notações formais; você deve escrever os requisitos de usuário em linguagem natural, com tabelas simples, formas e diagramas intuitivos.

Os requisitos de sistema são versões expandidas dos requisitos de usuário, usados por engenheiros de software como ponto de partida para o projeto do sistema. Eles acrescentam detalhes e explicam como os requisitos de usuário devem ser atendidos pelo sistema. Eles podem ser usados como parte do contrato para a implementação do sistema e devem consistir em uma especificação completa e detalhada de todo o sistema.

Os requisitos do sistema devem descrever apenas o comportamento externo do sistema e suas restrições operacionais. Eles não devem se preocupar com a forma como o sistema deve ser projetado ou implementado. No entanto, para atingir o nível de detalhamento necessário para especificar completamente um sistema de software complexo, é praticamente impossível eliminar todas as informações de projeto. Existem várias razões para isso:

1. Você pode precisar projetar uma arquitetura inicial do sistema para ajudar a estruturar a especificação de requisitos. Os requisitos de sistema são organizados de acordo com os diferentes subsistemas que compõem o sistema. Como discuto nos capítulos 6 e 18, essa definição da arquitetura é essencial caso você queira reusar componentes de software na implementação do sistema.
2. Na maioria dos casos, os sistemas devem interoperar com os sistemas existentes, que restringem o projeto e impõem requisitos sobre o novo sistema.
3. O uso de uma arquitetura específica para atender aos requisitos não funcionais (como programação N-version para alcançar a confiabilidade, discutida no Capítulo 13) pode ser necessário. Um regulador externo que precisa certificar que o sistema é seguro pode especificar que um projeto já certificado de arquitetura pode ser usado.

Os requisitos de usuário são quase sempre escritos em linguagem natural e suplementados no documento de requisitos por diagramas apropriados e tabelas. Requisitos de sistema também podem ser escritos em linguagem natural, mas também em outras notações com base em formulários, modelos gráficos ou matemáticos de sistema. A Tabela 4.3 resume as notações que poderiam ser usadas para escrever os requisitos de sistema.

Modelos gráficos são mais úteis quando você precisa mostrar como um estado se altera ou quando você precisa descrever uma sequência de ações. Diagramas de sequência e de estado da UML, descritos no Capítulo 5, mostram a sequência de ações que ocorrem em resposta a uma determinada mensagem ou evento. Especificações matemáticas formais são, por vezes, usadas para descrever os requisitos para os sistemas críticos de segurança ou de proteção, mas raramente são usadas em outras circunstâncias. Explico essa abordagem para elaboração de especificações no Capítulo 12.

4.3.1 Especificação em linguagem natural

Desde o início da engenharia de software a linguagem natural tem sido usada para escrever os requisitos para o software. É expressiva, intuitiva e universal. Também é potencialmente vaga, ambígua, e seu significado depende

Tabela 4.3 Formas de escrever uma especificação de requisitos de sistema.

Notação	Descrição
Sentenças em linguagem natural	Os requisitos são escritos em frases numeradas em linguagem natural. Cada frase deve expressar um requisito.
Linguagem natural estruturada	Os requisitos são escritos em linguagem natural em um formulário padrão ou *template*. Cada campo fornece informações sobre um aspecto do requisito.
Linguagem de descrição de projeto	Essa abordagem usa uma linguagem como de programação, mas com características mais abstratas, para especificar os requisitos, definindo um modelo operacional do sistema. Essa abordagem é pouco usada atualmente, embora possa ser útil para as especificações de interface.
Notações gráficas	Para definição dos requisitos funcionais para o sistema são usados modelos gráficos, suplementados por anotações de texto; diagramas de caso de uso e de sequência da UML são comumente usados.
Especificações matemáticas	Essas notações são baseadas em conceitos matemáticos, como máquinas de estado finito ou conjuntos. Embora essas especificações inequívocas possam reduzir a ambiguidade de um documento de requisitos, a maioria dos clientes não entende uma especificação formal. Eles não podem verificar que elas representam o que eles querem e são relutantes em aceitá-las como um contrato de sistema.

do conhecimento do leitor. Como resultado, tem havido muitas propostas de caminhos alternativos para a escrita dos requisitos. No entanto, nenhum destes tem sido amplamente adotado, e a linguagem natural continuará a ser a forma mais usada de especificação de requisitos do software e do sistema.

Para minimizar os mal-entendidos ao escrever requisitos em linguagem natural, recomendo algumas diretrizes simples:

1. Invente um formato-padrão e garanta que todas as definições de requisitos aderem a esse formato. A padronização do formato torna menos prováveis as omissões e mais fácil a verificação dos requisitos. O formato que eu uso expressa o requisito em uma única frase. Eu associo uma declaração com a justificativa para cada requisito de usuário para explicar por que o requisito foi proposto. A justificativa também pode incluir informações sobre quem propôs o requisito (a fonte do requisito), para saber quem consultar caso o requisito tenha de ser mudado.
2. Use uma linguagem consistente para distinguir entre os requisitos obrigatórios e os desejáveis. Os obrigatórios são requisitos aos quais o sistema tem de dar suporte e geralmente são escritos usando-se 'deve'. Requisitos desejáveis não são essenciais e são escritos usando-se 'pode'.
3. Use uma forma de destacar as partes fundamentais do requisito (negrito, itálico ou cores).
4. Não assuma que os leitores compreendem a linguagem técnica da engenharia de software. Frequentemente, palavras como 'arquitetura' e 'módulo' são mal interpretadas. Você deve, portanto, evitar o uso de jargões, siglas e acrônimos.
5. Sempre que possível, tente associar uma lógica a cada um dos requisitos de usuário. Essa justificativa deve explicar por que o requisito foi incluído, e é particularmente útil quando os requisitos são alterados, uma vez que pode ajudar a decidir sobre quais mudanças seriam indesejáveis.

O Quadro 4.2 ilustra como essas diretrizes podem ser usadas. Inclui dois requisitos para o software embutido para a bomba automática de insulina, citada no Capítulo 1. Você pode baixar a especificação de requisitos para a bomba de insulina completa das páginas do livro na Internet.

4.3.2 Especificações estruturadas

A linguagem natural estruturada é uma forma de escrever os requisitos do sistema na qual a liberdade do escritor dos requisitos é limitada e todos os requisitos são escritos em uma forma-padrão. Essa abordagem mantém grande parte da expressividade e compreensão da linguagem natural, mas garante certa uniformidade imposta sobre a especificação. Notações de linguagem estruturada usam *templates* para especificar os requisitos de sistema. A especificação pode usar construções de linguagem de programação para mostrar alternativas e iteração; além disso, pode destacar elementos-chave pelo uso de sombreamento ou fontes diferentes.

Os Robertson (ROBERTSON e ROBERTSON, 1999), em seu livro sobre o método VOLERE de engenharia de requisitos, recomendam que os requisitos de usuário sejam inicialmente escritos em cartões, um requisito por cartão. Eles sugerem um número de campos em cada cartão, algo como a lógica dos requisitos, as dependências de outros requisitos, a origem dos requisitos, materiais de apoio, e assim por diante. Essa é uma abordagem semelhante à do exemplo de uma especificação estruturada do Quadro 4.3.

Para usar uma abordagem estruturada para especificação de requisitos de sistema, você pode definir um ou mais *templates* para representar esses requisitos como formulários estruturados. A especificação pode ser estruturada em torno dos objetos manipulados pelo sistema, das funções desempenhadas pelo sistema, ou pelos eventos processados pelo sistema. Um exemplo de uma especificação baseada em formulários, nesse caso, que define a

Quadro 4.2 Exemplo de requisitos para o sistema de software de bomba de insulina.

3.2 O sistema deve medir o açúcar no sangue e fornecer insulina, se necessário, a cada dez minutos. (*Mudanças de açúcar no sangue são relativamente lentas, portanto, medições mais frequentes são desnecessárias; medições menos frequentes podem levar a níveis de açúcar desnecessariamente elevados.*)

3.6 O sistema deve, a cada minuto, executar uma rotina de autoteste com as condições a serem testadas e as ações associadas definidas na Quadro 4.3 (*A rotina de autoteste pode descobrir problemas de hardware e software e pode alertar o usuário para a impossibilidade de operar normalmente.*)

Quadro 4.3 Uma especificação estruturada de um requisito para uma bomba de insulina.

Bomba de insulina/Software de controle/SRS/3.3.2

Função	Calcula doses de insulina: nível seguro de açúcar.
Descrição	Calcula a dose de insulina a ser fornecida quando o nível de açúcar está na zona de segurança entre três e sete unidades.
Entradas	Leitura atual de açúcar (r2), duas leituras anteriores (r0 e r1).
Fonte	Leitura atual da taxa de açúcar pelo sensor. Outras leituras da memória.
Saídas	CompDose — a dose de insulina a ser fornecida.
Destino	*Loop* principal de controle.
Ação	CompDose é zero se o nível de açúcar está estável ou em queda ou se o nível está aumentando, mas a taxa de aumento está diminuindo. Se o nível está aumentando e a taxa de aumento está aumentando, então CompDose é calculado dividindo-se a diferença entre o nível atual de açúcar e o nível anterior por quatro e arredondando-se o resultado. Se o resultado é arredondado para zero, então CompDose é definida como a dose mínima que pode ser fornecida.
Requisitos	Duas leituras anteriores, de modo que a taxa de variação do nível de açúcar pode ser calculada.
Pré-condição	O reservatório de insulina contém, no mínimo, o máximo de dose única permitida de insulina.
Pós-condições	r0 é substituída por r1 e r1 é substituída por r2.
Efeitos colaterais	Nenhum.

forma de calcular a dose de insulina a ser fornecida quando o açúcar no sangue está dentro de uma faixa de segurança, é mostrado no Quadro 4.3

Quando um formulário-padrão é usado para especificar requisitos funcionais, as seguintes informações devem ser incluídas:

1. A descrição da função ou entidade a ser especificada.
2. Uma descrição de suas entradas e de onde elas vieram.
3. Uma descrição de suas saídas e para onde elas irão.
4. Informações sobre a informação necessária para o processamento ou outras entidades usadas no sistema (a parte '*requires*').
5. Uma descrição da ação a ser tomada.
6. Se uma abordagem funcional é usada, uma pré-condição define o que deve ser verdade antes que a função seja chamada, e é chamada uma pós-condição, especificando o que é verdade depois da função.
7. Uma descrição dos efeitos colaterais da operação (caso haja algum).

Usar as especificações estruturadas elimina alguns dos problemas de especificação em linguagem natural. Reduz-se a variabilidade na especificação, e os requisitos são organizados de forma mais eficaz. No entanto, algumas vezes ainda é difícil escrever os requisitos de forma clara e inequívoca, especialmente quando processamentos complexos (como, por exemplo, calcular a dose de insulina) devem ser especificados.

Para resolver esse problema, você pode adicionar informações extras aos requisitos em linguagem natural, usando tabelas ou modelos gráficos do sistema, por exemplo. Estes podem mostrar como os cálculos são executados, como o sistema muda de estado, como os usuários interagem com o sistema e como sequências de ações são executadas.

As tabelas são particularmente úteis quando há um número de situações alternativas possíveis e é necessário descrever as ações a serem tomadas para cada uma delas. A bomba de insulina baseia seus cálculos na necessidade de insulina sobre a taxa de variação dos níveis de açúcar no sangue. As taxas de variação são calculadas usando as leituras atuais e anteriores. A Tabela 4.4 é uma descrição tabular de como a taxa de variação de açúcar no sangue é usada para calcular a quantidade de insulina a ser fornecida.

Tabela 4.4 Especificação tabular de processamento para uma bomba de insulina.

Condição	Ação
Nível de açúcar diminuindo (r2 < r1)	CompDose = 0
Nível de açúcar estável (r2 = r1)	CompDose = 0
Nível de açúcar aumentando e a taxa de aumento decrescente [(r2 – r1) < (r1 – r0)]	CompDose = 0
Nível de açúcar aumentando e a taxa de aumento estável ou crescente [(r2 – r1) ≥ (r1 – r0)]	CompDose = arredondar [(r2 – r1) / 4)]. Se o resultado arredondado = 0, então CompDose = MinimumDose

4.4 Processos de engenharia de requisitos

Como discutido no Capítulo 2, os processos de engenharia de requisitos podem incluir quatro atividades de alto nível. Elas visam avaliar se o sistema é útil para a empresa (estudo de viabilidade), descobrindo requisitos (elicitação e análise), convertendo-os em alguma forma-padrão (especificação), e verificar se os requisitos realmente definem o sistema que o cliente quer (validação). Mostrei essas atividades como processos sequenciais na Figura 2.6. No entanto, na prática, a engenharia de requisitos é um processo iterativo em que as atividades são intercaladas.

A Figura 4.4 mostra esta intercalação. As atividades são organizadas em torno de uma espiral, como um processo iterativo, sendo a saída um documento de requisitos de sistema. A quantidade de tempo e esforço dedicados a cada atividade em cada iteração depende do estágio do processo como um todo e do tipo de sistema que está sendo desenvolvido. No início do processo, o esforço maior será a compreensão dos requisitos de negócio e não funcionais em alto nível, bem como dos requisitos de usuário para o sistema. Mais tarde no processo, nos anéis externos da espiral, o esforço maior será dedicado a elicitar e compreender os requisitos de sistema em detalhes.

Esse modelo espiral acomoda abordagens em que os requisitos são desenvolvidos em diferentes níveis de detalhamento. O número de iterações em torno da espiral pode variar; assim, a espiral pode acabar depois da definição de alguns ou de todos os requisitos de usuário. No lugar de prototipação, o desenvolvimento ágil pode ser usado para que os requisitos e a implementação do sistema sejam desenvolvidos em conjunto.

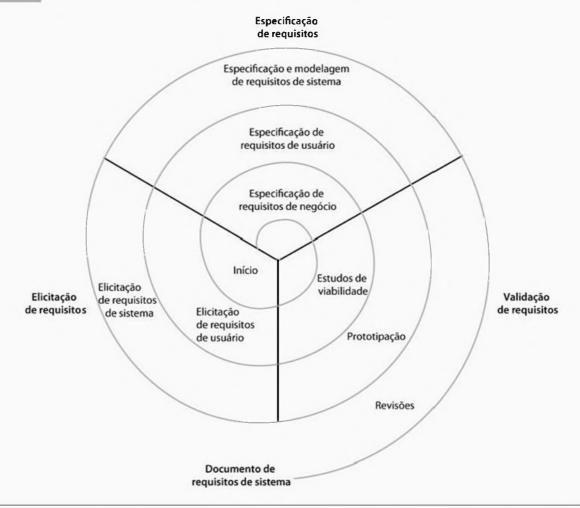
Algumas pessoas consideram a engenharia de requisitos o processo de aplicação de um método de análise estruturada, como a análise orientada a objetos (LARMAN, 2002). Trata-se de analisar o sistema e desenvolver um conjunto de modelos gráficos de sistema, como modelos de casos de uso, que, então, servem como uma especificação do sistema. O conjunto de modelos descreve o comportamento do sistema e é anotado com informações adicionais, descrevendo, por exemplo, o desempenho ou a confiabilidade requerida do sistema.

Embora os métodos estruturados tenham um papel a desempenhar no processo de engenharia de requisitos, existe muito mais para a engenharia de requisitos do que o que é coberto por esses métodos. Elicitação de requisitos, em particular, é uma atividade centrada em pessoas, e as pessoas não gostam de restrições impostas por modelos rígidos de sistema.

Em praticamente todos os sistemas os requisitos mudam. As pessoas envolvidas desenvolvem uma melhor compreensão do que querem do software, a organização que compra o sistema também muda, modificações são feitas no hardware, no software e no ambiente organizacional do sistema. O processo de gerenciamento desses requisitos em constante mudança é chamado gerenciamento de requisitos, sobre o qual eu escrevo na Seção 4.7.

4.5 Elicitação e análise de requisitos

Após um estudo inicial de viabilidade, o próximo estágio do processo de engenharia de requisitos é a elicitação e análise de requisitos. Nessa atividade, os engenheiros de software trabalham com clientes e usuários finais do sistema para obter informações sobre o domínio da aplicação, os serviços que o sistema deve oferecer, o desempenho do sistema, restrições de hardware e assim por diante.

Figura 4.5 Uma visão em espiral do processo de engenharia de requisitos.

A elicitação e análise de requisitos podem envolver diversos tipos de pessoas em uma organização. Um *stakeholder* do sistema é quem tem alguma influência direta ou indireta sobre os requisitos do sistema. Os *stakeholders* incluem os usuários finais que irão interagir com o sistema e qualquer outra pessoa em uma organização que será afetada por ele. Outros *stakeholders* do sistema podem ser os engenheiros que estão desenvolvendo ou mantendo outros sistemas relacionados a esse, como também gerentes de negócios, especialistas de domínio e representantes sindicais.

Um modelo do processo de elicitação e análise é mostrado na Figura 4.6. Cada organização terá sua própria versão ou instância desse modelo geral, dependendo de fatores locais, como a *expertise* do pessoal, o tipo de sistema a ser desenvolvido, as normas usadas etc.

As atividades do processo são:

1. *Descoberta de requisitos.* Essa é a atividade de interação com os *stakeholders* do sistema para descobrir seus requisitos. Os requisitos de domínio dos *stakeholders* e da documentação também são descobertos durante essa atividade. Existem várias técnicas complementares que podem ser usadas para descoberta de requisitos, que discuto mais adiante.
2. *Classificação e organização de requisitos.* Essa atividade toma a coleção de requisitos não estruturados, agrupa requisitos relacionados e os organiza em grupos coerentes. A forma mais comum de agrupar os requisitos é o uso de um modelo de arquitetura do sistema para identificar subsistemas e associar requisitos a cada subsistema. Na prática, a engenharia de requisitos e projeto da arquitetura não podem ser atividades completamente separadas.
3. *Priorização e negociação de requisitos.* Inevitavelmente, quando os vários *stakeholders* estão envolvidos, os requisitos entram em conflito. Essa atividade está relacionada com a priorização de requisitos e em encontrar e

Figura 4.6 O processo de elicitação e análise de requisitos.

resolver os conflitos por meio da negociação de requisitos. Normalmente, os *stakeholders* precisam se encontrar para resolver as diferenças e chegar a um acordo sobre os requisitos.

4. *Especificação de requisitos*. Os requisitos são documentados e inseridos no próximo ciclo da espiral. Documentos formais ou informais de requisitos podem ser produzidos, como discutido na Seção 4.3.

A Figura 4.5 mostra que a elicitação e análise de requisitos é um processo iterativo, com *feedback* contínuo de cada atividade para as outras atividades. O ciclo do processo começa com a descoberta de requisitos e termina com sua documentação. O entendimento do analista de requisitos melhora a cada rodada do ciclo. Quando se completa o documento de requisitos, o ciclo termina.

Elicitar e compreender os requisitos dos *stakeholders* do sistema é um processo difícil por várias razões:

1. Exceto em termos gerais, os *stakeholders* costumam não saber o que querem de um sistema computacional; eles podem achar difícil articular o que querem que o sistema faça, e, como não sabem o que é viável e o que não é, podem fazer exigências inviáveis.
2. Naturalmente, os *stakeholders* expressam requisitos em seus próprios termos e com o conhecimento implícito de seu próprio trabalho. Engenheiros de requisitos, sem experiência no domínio do cliente, podem não entender esses requisitos.
3. Diferentes *stakeholders* têm requisitos diferentes e podem expressar essas diferenças de várias maneiras. Engenheiros de requisitos precisam descobrir todas as potenciais fontes de requisitos e descobrir as semelhanças e conflitos.
4. Fatores políticos podem influenciar os requisitos de um sistema. Os gerentes podem exigir requisitos específicos, porque estes lhes permitirão aumentar sua influência na organização.
5. O ambiente econômico e empresarial no qual a análise ocorre é dinâmico. É inevitável que ocorram mudanças durante o processo de análise. A importância dos requisitos específicos pode mudar. Novos requisitos podem surgir a partir de novos *stakeholders* que não foram inicialmente consultados.

Inevitavelmente, os diferentes *stakeholders* têm opiniões diferentes sobre a importância e prioridade dos requisitos e, por vezes, essas opiniões são conflitantes. Durante o processo, você deve organizar as negociações regulares dos *stakeholders*, de modo que os compromissos possam ser cumpridos. É completamente impossível satisfazer a todos os *stakeholders*, mas, se alguns deles perceberem que suas opiniões não foram devidamente consideradas, poderão, deliberadamente, tentar arruinar o processo de RE.

No estágio de especificação de requisitos, aqueles que foram elicitados até esse momento são documentados de forma a ajudar na descoberta de novos requisitos. Nesse estágio, uma versão inicial do documento de requisitos do sistema pode ser produzida com seções faltantes e requisitos incompletos. Como alternativa, os requisitos podem ser documentados de uma forma completamente diferente (por exemplo, em uma planilha ou em cartões). Escrever os requisitos em cartões pode ser muito eficaz, pois são fáceis para os *stakeholders* lidarem, mudarem e organizarem.

4.5.1 Descoberta de requisitos

A descoberta de requisitos (às vezes, chamada elicitação de requisitos) é o processo de reunir informações sobre o sistema requerido e os sistemas existentes e separar dessas informações os requisitos de usuário e de sistema. Fontes de informação durante a fase de descoberta de requisitos incluem documentação, *stakeholders* do sistema e especificações de sistemas similares. Você interage com os *stakeholders* por meio da observação e de entrevistas e pode usar cenários e protótipos para ajudar os *stakeholders* a compreenderem o que o sistema vai ser.

Os *stakeholders* variam desde os usuários finais, passando pelos gerentes do sistema até *stakeholders* externos, como reguladores, que certificam a aceitabilidade do sistema. Por exemplo, os *stakeholders* do sistema de informação da saúde mental de pacientes incluem:

1. Os pacientes cujas informações estão registradas no sistema.
2. Os médicos responsáveis pela avaliação e tratamento dos pacientes.
3. Os enfermeiros que, alinhados com os médicos, coordenam as consultas e administram tratamentos.
4. As (os) recepcionistas dos médicos, que gerenciam as consultas dos pacientes.
5. A equipe de TI, responsável pela instalação e manutenção do sistema.
6. Um gerente de ética médica, que deve garantir que o sistema atenda às diretrizes éticas atuais no atendimento ao paciente.
7. Gerentes de saúde, que obtêm informações de gerenciamento a partir do sistema.
8. A equipe de registros médicos, responsável por garantir que as informações do sistema sejam mantidas e preservadas, e que os procedimentos de manutenção dos registros sejam devidamente implementados.

Além dos *stakeholders* do sistema, já vimos que os requisitos também podem vir a partir do domínio da aplicação e de outros sistemas que interagem com o sistema especificado. Durante o processo de elicitação de requisitos, todos esses devem ser considerados.

Essas diferentes fontes de requisitos (*stakeholders*, domínio, sistemas) podem ser representadas como pontos de vista do sistema, com cada ponto de vista mostrando um subconjunto dos requisitos para o sistema. Pontos de vista diferentes sobre um problema percebem o problema de maneiras diferentes. No entanto, suas perspectivas não são completamente independentes; em geral, elas se sobrepõem e, dessa forma, apresentam requisitos comuns. Você pode usar esses pontos de vista para estruturar a descoberta e a documentação dos requisitos do sistema.

4.5.2 Entrevistas

Entrevistas formais ou informais com os *stakeholders* do sistema são parte da maioria dos processos de engenharia de requisitos. Nessas entrevistas, a equipe de engenharia de requisitos questiona os *stakeholders* sobre o sistema que usam no momento e sobre o sistema que será desenvolvido. Requisitos surgem a partir das respostas a essas perguntas. As entrevistas podem ser de dois tipos:

1. Entrevistas fechadas, em que o *stakeholder* responde a um conjunto predefinido de perguntas.
2. Entrevistas abertas, em que não existe uma agenda predefinida. A equipe de engenharia de requisitos explora uma série de questões com os *stakeholders* do sistema e, assim, desenvolve uma melhor compreensão de suas necessidades.

Na prática, as entrevistas com os *stakeholders* costumam ser uma mistura de ambos os tipos. Você poderá ter de obter a resposta a determinadas questões, mas é comum que estas levem a outras, discutidas de forma menos estruturada. Discussões totalmente abertas raramente funcionam bem. Você geralmente tem de fazer algumas perguntas para começar e manter a entrevista centrada no sistema que será desenvolvido.

Entrevistas são boas para obter uma compreensão global sobre o que os *stakeholders* fazem, como eles podem interagir com o novo sistema e as dificuldades que eles enfrentam com os sistemas atuais. As pessoas gostam de falar sobre seus trabalhos e geralmente ficam felizes de se envolver em entrevistas. No entanto, as entrevistas não são tão úteis na compreensão dos requisitos do domínio da aplicação.

Pode ser difícil elicitar conhecimento do domínio por meio de entrevistas por duas razões:

1. Todos os especialistas em aplicações usam terminologias e jargões específicos para um domínio. Para eles, é impossível discutir os requisitos de domínio essa terminologia. Eles normalmente usam a terminologia de forma precisa e sutil, o que dificulta a compreensão dos engenheiros de requisitos.
2. O conhecimento de domínio é tão familiar aos *stakeholders* que eles têm dificuldade de explicá-lo, ou pensam que é tão fundamental que não vale a pena mencionar. Por exemplo, para um bibliotecário, não é necessário dizer que todas as aquisições são catalogadas antes de serem adicionadas à biblioteca. No entanto, isso pode não ser óbvio para o entrevistador, e acaba não sendo levado em conta nos requisitos.

Entrevistas também não são uma técnica eficaz para a elicitação do conhecimento sobre os requisitos e restrições organizacionais, porque, entre as diferentes pessoas da organização, existem sutis relações de poder. As estruturas organizacionais publicadas raramente correspondem à realidade do processo de tomada de decisão em uma organização, mas os entrevistados podem preferir não revelar a estrutura real, e sim a estrutura teórica, para um estranho. Em geral, a maioria das pessoas é relutante em discutir questões políticas e organizacionais que podem afetar os requisitos.

Entrevistadores eficazes têm duas características:

1. Eles estão abertos a novas ideias, evitam ideias preconcebidas sobre os requisitos e estão dispostos a ouvir os *stakeholders*. Mesmo que o *stakeholder* apresente requisitos-surpresa, eles estão dispostos a mudar de ideia sobre o sistema.
2. Eles estimulam o entrevistado a participar de discussões com uma questão-trampolim, uma proposta de requisitos ou trabalhando em conjunto em um protótipo do sistema. É improvável que dizer às pessoas "diga-me o que quiser" resulte em informações úteis. É muito mais fácil falar em um contexto definido do que em termos gerais.

Informações recolhidas em entrevistas suplementam outras informações sobre o sistema, advindas de documentos que descrevem processos de negócios ou sistemas existentes, observações do usuário etc. Em alguns casos, além da informação contida nos documentos do sistema, as entrevistas podem ser a única fonte de informação sobre os requisitos do sistema. No entanto, a entrevista por si só pode deixar escapar informações essenciais; por isso, deve ser usada em conjunto com outras técnicas de elicitação de requisitos.

4.5.3 Cenários

As pessoas geralmente acham mais fácil se relacionar com exemplos da vida real do que com descrições abstratas. Elas podem compreender e criticar um cenário de como elas podem interagir com um sistema de software. Engenheiros de requisitos podem usar a informação obtida a partir deste debate para formular os requisitos do sistema atual.

Os cenários podem ser particularmente úteis para adicionar detalhes a uma descrição geral de requisitos. Trata-se de descrições de exemplos de sessões de interação. Cada cenário geralmente cobre um pequeno número de interações possíveis. Diferentes cenários são desenvolvidos e oferecem diversos tipos de informação em variados níveis de detalhamento sobre o sistema. As estórias usadas em Extreme Programming, discutidas no Capítulo 3, são um tipo de cenário de requisitos.

Um cenário começa com um esboço da interação. Durante o processo de elicitação, são adicionados detalhes ao esboço, para criar uma descrição completa dessa interação. Em sua forma mais geral, um cenário pode incluir:

1. Uma descrição do que o sistema e os usuários esperam quando o cenário se iniciar.
2. Uma descrição do fluxo normal de eventos no cenário.
3. Uma descrição do que pode dar errado e como isso é tratado.
4. Informações sobre outras atividades que podem acontecer ao mesmo tempo.
5. Uma descrição do estado do sistema quando o cenário acaba.

A elicitação baseada em cenários envolve o trabalho com os *stakeholders* para identificar cenários e capturar detalhes que serão incluídos nesses cenários. Os cenários podem ser escritos como texto, suplementados por diagramas, telas etc. Outra possibilidade é uma abordagem mais estruturada, em que cenários de eventos ou casos de uso podem ser usados.

Como exemplo de um cenário de texto simples, considere como o MHC-PMS pode ser usado para introduzir dados de um novo paciente (Quadro 4.4). Quando um novo paciente vai a uma clínica, um novo registro é criado

Quadro 4.4 Cenário para a coleta do histórico médico em MHC-PMS.

Suposição inicial:
O paciente é atendido em uma clínica médica por uma recepcionista; ela gera um registro no sistema e coleta suas informações pessoais (nome, endereço, idade etc.). Uma enfermeira é conectada ao sistema e coleta o histórico médico do paciente.

Normal:
A enfermeira busca o paciente pelo sobrenome. Se houver mais de um paciente com o mesmo sobrenome, o nome e a data de nascimento são usados para identificar o paciente.
A enfermeira escolhe a opção do menu para adicionar o histórico médico.
A enfermeira segue, então, uma série de *prompts* do sistema para inserir informações sobre consultas em outros locais, os problemas de saúde mental (entrada de texto livre), condições médicas (enfermeira seleciona condições do menu), medicação atual (selecionado no menu), alergias (texto livre) e informações da vida doméstica (formulário).

O que pode dar errado:
O prontuário do paciente não existe ou não pôde ser encontrado. A enfermeira deve criar um novo registro e registrar as informações pessoais.
As condições do paciente ou a medicação em uso não estão inscritas no menu. A enfermeira deve escolher a opção 'outros' e inserir texto livre com descrição da condição/medicação.
O paciente não pode/não fornecerá informações sobre seu histórico médico. A enfermeira deve inserir um texto livre registrando a incapacidade/relutância do paciente em fornecer as informações. O sistema deve imprimir o formulário-padrão de exclusão afirmando que a falta de informação pode significar que o tratamento será limitado ou postergado. Este deverá ser assinado e entregue ao paciente.

Outras atividades:
Enquanto a informação está sendo inserida, o registro pode ser consultado, mas não editado por outros agentes.

Estado do sistema na conclusão:
O usuário está conectado. O prontuário do paciente, incluindo seu histórico médico, é inserido no banco de dados e um registro é adicionado ao *log* do sistema, mostrando o tempo de início e fim da sessão e a enfermeira envolvida.

por uma recepcionista, e suas informações pessoais (nome, idade etc.) são acrescentadas nessa ficha. Em seguida, uma enfermeira entrevista o paciente e coleta seu histórico médico. Então, o paciente faz uma consulta inicial com o médico, que faz um diagnóstico e, se for o caso, recomenda um tratamento. O cenário mostra o que acontece quando o histórico médico é coletado.

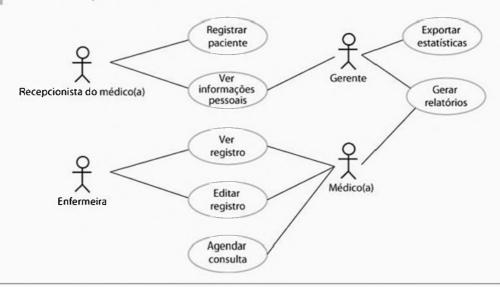
4.5.4 Casos de uso

Os casos de uso são uma técnica de descoberta de requisitos introduzida inicialmente no método Objectory (JACOBSON et al., 1993). Eles já se tornaram uma característica fundamental da linguagem de modelagem unificada (UML — do inglês *unified modeling language*). Em sua forma mais simples, um caso de uso identifica os atores envolvidos em uma interação e dá nome ao tipo de interação. Essa é, então, suplementada por informações adicionais que descrevem a interação com o sistema. A informação adicional pode ser uma descrição textual ou um ou mais modelos gráficos, como diagrama de sequência ou de estados da UML.

Os casos de uso são documentados por um diagrama de casos de uso de alto nível. O conjunto de casos de uso representa todas as possíveis interações que serão descritas nos requisitos de sistema. Atores, que podem ser pessoas ou outros sistemas, são representados como figuras 'palito'. Cada classe de interação é representada por uma elipse. Linhas fazem a ligação entre os atores e a interação. Opcionalmente, pontas de flechas podem ser adicionadas às linhas para mostrar como a interação se inicia. Essa situação é ilustrada na Figura 4.6, que mostra alguns dos casos de uso para o sistema de informações de pacientes.

Não há distinção entre cenários e casos de uso que seja simples e rápida. Algumas pessoas consideram cada caso de uso um cenário único; outros, como sugerido por Stevens e Pooley (2006), encapsulam um conjunto de cenários em um único caso de uso. Cada cenário é um segmento através do caso de uso. Portanto, seria um cenário para a interação normal além de cenários para cada possível exceção. Você pode, na prática, usá-los de qualquer forma.

Os casos de uso identificam as interações individuais entre o sistema e seus usuários ou outros sistemas. Cada caso de uso deve ser documentado com uma descrição textual. Esta, por sua vez, pode ser ligada a outros modelos

Figura 4.6 Casos de uso para o MHC-PMS.

UML que desenvolverão o cenário com mais detalhes. Por exemplo, uma breve descrição do caso de uso Agendar consulta, representado na Figura 4.6, pode ser:

> *Agendar a consulta permite que dois ou mais médicos de consultórios diferentes possam ler o mesmo registro ao mesmo tempo. Um médico deve escolher, em um menu de lista de médicos on-line, as pessoas envolvidas. O prontuário do paciente é então exibido em suas telas, mas apenas o primeiro médico pode editar o registro. Além disso, uma janela de mensagens de texto é criada para ajudar a coordenar as ações. Supõe-se que uma conferência telefônica para comunicação por voz será estabelecida separadamente.*

Cenários e casos de uso são técnicas eficazes para elicitar requisitos dos *stakeholders* que vão interagir diretamente com o sistema. Cada tipo de interação pode ser representado como um caso de uso. No entanto, devido a seu foco nas interações com o sistema, eles não são tão eficazes para elicitar restrições ou requisitos de negócios e não funcionais em alto nível ou para descobrir requisitos de domínio.

A UML é, de fato, um padrão para a modelagem orientada a objetos, e assim, casos de uso e elicitação baseada em casos de uso são amplamente usados para a elicitação de requisitos. Mais adiante, no Capítulo 5, discuto casos de uso e mostro como eles são usados, juntamente com outros modelos, para documentar um projeto de sistema.

4.5.5 Etnografia

Os sistemas de software não existem isoladamente. Eles são usados em um contexto social e organizacional, e requisitos de software do sistema podem ser derivados ou restringidos desse contexto. Geralmente, satisfazer a esses requisitos sociais e organizacionais é crítico para o sucesso do sistema. Uma razão pela qual muitos sistemas de software são entregues, mas nunca são usados, é que seus requisitos não levam devidamente em conta a forma como o contexto social e organizacional afeta o funcionamento prático do sistema.

Etnografia é uma técnica de observação que pode ser usada para compreender os processos operacionais e ajudar a extrair os requisitos de apoio para esses processos. Um analista faz uma imersão no ambiente de trabalho em que o sistema será usado. O trabalho do dia a dia é observado e são feitas anotações sobre as tarefas reais em que os participantes estão envolvidos. O valor da etnografia é que ela ajuda a descobrir requisitos implícitos do sistema que refletem as formas reais com que as pessoas trabalham, em vez de refletir processos formais definidos pela organização.

Frequentemente, as pessoas acham muito difícil expressar os detalhes de seu trabalho, pois isso é secundário para elas. Elas entendem o próprio trabalho, mas não compreendem sua relação com outros trabalhos na organização. Fatores sociais e organizacionais que afetam o trabalho, mas que não são óbvios para os indivíduos, podem ficar claros apenas quando analisados por um observador imparcial. Por exemplo, um grupo de trabalho pode se auto-organizar de maneira que os membros conheçam mutuamente seus trabalhos e, em caso de ausência de algum deles, possam dividir as tarefas. Essa informação pode não ser mencionada durante uma entrevista, pois o grupo não a percebe como parte integrante de seu trabalho.

Suchman (1987) foi pioneira no uso da etnografia na análise do trabalho de escritório. Ela constatou que as práticas de trabalho eram muito mais ricas, mais complexas e mais dinâmicas do que os modelos simples assumidos pelos sistemas de automação de escritório. A diferença entre o trabalho presumido e o real foi a razão mais importante por que esses sistemas de escritório não causaram efeitos significativos sobre a produtividade. Crabtree (2003) discute uma ampla gama de estudos desde então e descreve, em geral, o uso da etnografia em projeto de sistemas. Em minha pesquisa, tenho investigado métodos de integração de etnografia no processo de engenharia de software por meio de sua ligação com os métodos de engenharia de requisitos (VILLER e SOMMERVILLE, 1999; VILLER e SOMMERVILLE, 2000) e documentação de padrões de interação em sistemas cooperativos (MARTIN et al., 2001; MARTIN et al., 2002; MARTIN e SOMMERVILLE, 2004).

A etnografia é particularmente eficaz para descobrir dois tipos de requisitos:

1. Requisitos derivados da maneira como as pessoas realmente trabalham, e não da forma como as definições dos processos dizem que deveriam trabalhar. Por exemplo, controladores de tráfego aéreo podem desligar um sistema de alerta de conflitos que detecta aeronaves com rotas em colisão, embora os procedimentos de controle normal especifiquem que ele deve ser usado. Eles deliberadamente colocam a aeronave em caminhos conflitantes, por um curto período, para ajudar no gerenciamento do espaço aéreo. Sua estratégia de controle é projetada para assegurar que os aviões sejam afastados dessa rota conflitante antes que surjam problemas, e eles acham que o alarme de alerta distrai seu trabalho.
2. Requisitos derivados da cooperação e conhecimento das atividades de outras pessoas. Por exemplo, controladores de tráfego aéreo podem usar conhecimento do trabalho de outros controladores para prever o número de aeronaves que entrarão em seu setor de controle. Eles, então, modificam suas estratégias de controle, dependendo do volume de trabalho previsto. Portanto, um sistema ATC automatizado deve permitir aos controladores de um setor alguma visibilidade do trabalho em setores adjacentes.

A etnografia pode ser combinada com prototipação (Figura 4.7). A etnografia informa o desenvolvimento do protótipo, para que menos ciclos de refinamento do protótipo sejam necessários. Além disso, a prototipação dá foco para a etnografia, ao identificar problemas e questões que podem ser discutidos com o etnógrafo. Esse profissional deve procurar as respostas para essas perguntas durante a próxima fase do estudo do sistema (SOMMERVILLE et al., 1993).

Estudos etnográficos podem revelar detalhes críticos de processo que, muitas vezes, são ignorados por outras técnicas de elicitação de requisitos. Contudo, uma vez que o foco é o usuário final, essa abordagem nem sempre é apropriada para descobrir requisitos organizacionais ou de domínio. Eles nem sempre podem identificar novos recursos que devem ser adicionados ao sistema. A etnografia, portanto, não é uma abordagem completa para elicitação e deve ser usada para complementar outras abordagens, como análise de casos de uso.

4.6 Validação de requisitos

A validação de requisitos é o processo pelo qual se verifica se os requisitos definem o sistema que o cliente realmente quer. Ela se sobrepõe à análise, uma vez que está preocupada em encontrar problemas com os requisitos. A validação de requisitos é importante porque erros em um documento de requisitos podem gerar altos custos de retrabalho quando descobertos durante o desenvolvimento ou após o sistema já estar em serviço.

O custo para consertar um problema de requisitos por meio de uma mudança no sistema é geralmente muito maior do que o custo de consertar erros de projeto ou de codificação. A razão para isso é que a ocorrência de mudança dos requisitos normalmente significa que o projeto e a implementação do sistema também devem ser alterados. Além disso, o sistema deve, posteriormente, ser retestado.

Figura 4.7 Etnografia e prototipação para análise de requisitos.

Durante o processo de validação de requisitos, diferentes tipos de verificação devem ser efetuados com os requisitos no documento de requisitos. Essas verificações incluem:

1. *Verificações de validade*. Um usuário pode pensar que é necessário um sistema para executar determinadas funções. No entanto, maior reflexão e análise mais aprofundada podem identificar funções necessárias, adicionais ou diferentes. Os sistemas têm diversos *stakeholders* com diferentes necessidades, e qualquer conjunto de requisitos é inevitavelmente um compromisso da comunidade de *stakeholders*.
2. *Verificações de consistência*. Requisitos no documento não devem entrar em conflito. Ou seja, não deve haver restrições contraditórias ou descrições diferentes da mesma função do sistema.
3. *Verificações de completude*. O documento de requisitos deve incluir requisitos que definam todas as funções e as restrições pretendidas pelo usuário do sistema.
4. *Verificações de realismo*. Usando o conhecimento das tecnologias existentes, os requisitos devem ser verificados para assegurar que realmente podem ser implementados. Essas verificações devem considerar o orçamento e o cronograma para o desenvolvimento do sistema.
5. *Verificabilidade*. Para reduzir o potencial de conflito entre o cliente e o contratante, os requisitos do sistema devem ser passíveis de verificação. Isso significa que você deve ser capaz de escrever um conjunto de testes que demonstrem que o sistema entregue atende a cada requisito especificado.

Existe uma série de técnicas de validação de requisitos que podem ser usadas individualmente ou em conjunto:

1. *Revisões de requisitos*. Os requisitos são analisados sistematicamente por uma equipe de revisores que verifica erros e inconsistências.
2. *Prototipação*. Nessa abordagem para validação, um modelo executável do sistema em questão é demonstrado para os usuários finais e clientes. Estes podem experimentar o modelo para verificar se ele atende a suas reais necessidades.
3. *Geração de casos de teste*. Os requisitos devem ser testáveis. Se os testes forem concebidos como parte do processo de validação, isso frequentemente revela problemas de requisitos. Se é difícil ou impossível projetar um teste, isso normalmente significa que os requisitos serão difíceis de serem implementados e devem ser reconsiderados. O desenvolvimento de testes a partir dos requisitos do usuário antes de qualquer código ser escrito é parte integrante do Extreme Programming.

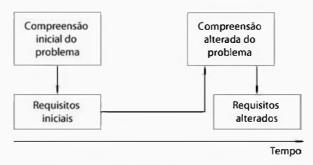
Você não deve subestimar os problemas envolvidos na validação de requisitos. Afinal, é difícil mostrar que um conjunto de requisitos atende de fato às necessidades de um usuário; os usuários precisam imaginar o sistema em operação e como esse sistema se encaixaria em seu trabalho. Até para profissionais qualificados de informática é difícil fazer esse tipo de análise abstrata, e é mais difícil ainda para os usuários do sistema. Como resultado, durante o processo de validação dos requisitos você raramente encontrará todos os problemas de requisitos. Após o ajuste do documento de requisitos, é inevitável a necessidade de mudanças nos requisitos para corrigir omissões e equívocos.

4.7 Gerenciamento de requisitos

Os requisitos para sistemas de software de grande porte estão sempre mudando. Uma razão para isso é que esses sistemas geralmente são desenvolvidos para enfrentar os problemas 'maus' — problemas que não podem ser completamente definidos. Porque o problema não pode ser totalmente definido, os requisitos de software são obrigados a ser incompletos. Durante o processo de software, o entendimento dos *stakeholders* a respeito do problema está em constante mutação (Figura 4.8). Logo, os requisitos de sistema devem evoluir para refletir essas novas percepções do problema.

Uma vez que um sistema tenha sido instalado e seja usado regularmente, inevitavelmente surgirão novos requisitos. É difícil para os usuários e clientes do sistema anteciparem os efeitos que o novo sistema terá sobre seus processos de negócio e sobre a forma que o trabalho é realizado. Quando os usuários finais tiverem a experiência de um sistema, descobrirão novas necessidades e prioridades. Existem várias razões pelas quais a mudança é inevitável:

1. Após a instalação, o ambiente técnico e de negócios do sistema sempre muda. Um novo hardware pode ser introduzido, pode ser necessário fazer a interface do sistema com outros sistemas, as prioridades do negócio podem mudar (com consequentes alterações necessárias no apoio do sistema), podem ser introduzidas novas legislações e regulamentos aos quais o sistema deve, necessariamente, respeitar etc.

Figura 4.8 Evolução dos requisitos.

2. As pessoas que pagam por um sistema e os usuários desse sistema raramente são os mesmos. Clientes do sistema impõem requisitos devido a restrições orçamentárias e organizacionais, os quais podem entrar em conflito com os requisitos do usuário final, e, após a entrega, novos recursos podem ser adicionados, para dar suporte ao usuário, a fim de que o sistema cumpra suas metas.
3. Geralmente, sistemas de grande porte têm uma comunidade de diversos usuários, com diferentes requisitos e prioridades, que podem ser conflitantes ou contraditórios. Os requisitos do sistema final são, certamente, um compromisso entre esses usuários, e, com a experiência, frequentemente se descobre que o balanço de apoio prestado aos diferentes usuários precisa ser mudado.

O gerenciamento de requisitos é o processo de compreensão e controle das mudanças nos requisitos do sistema. Você precisa se manter a par das necessidades individuais e manter as ligações entre as necessidades dependentes para conseguir avaliar o impacto das mudanças nos requisitos. Você precisa estabelecer um processo formal para fazer propostas de mudanças e a ligação destas às exigências do sistema. O processo formal de gerenciamento de requisitos deve começar assim que uma versão preliminar do documento de requisitos estiver disponível. No entanto, você deve começar a planejar como gerenciar mudanças de requisitos durante o processo de elicitação de requisitos.

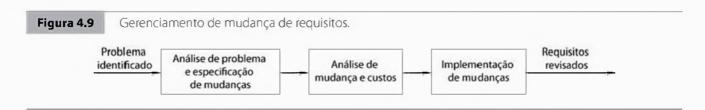
4.7.1 Planejamento de gerenciamento de requisitos

O planejamento é o primeiro estágio essencial no processo de gerenciamento de requisitos, e determina o nível de detalhamento requerido no gerenciamento de requisitos. Durante o estágio de gerenciamento de requisitos, você deve decidir sobre:

1. *Identificação de requisitos*. Cada requisito deve ser identificado unicamente para poder ser comparado com outros requisitos e usado em avaliações de rastreabilidade.
2. *Processo de gerenciamento de mudanças*. Esse é o conjunto de atividades que avaliam o impacto e o custo das mudanças. Na próxima seção, vou discutir detalhadamente esse processo.
3. *Políticas de rastreabilidade*. Definem os relacionamentos entre cada requisito e entre os requisitos e o projeto de sistema que deve ser registrado. A política de rastreabilidade também deve definir como esses registros devem ser mantidos.
4. *Ferramenta de apoio*. Gerenciamento de requisitos envolve o processamento de grandes quantidades de informações sobre os requisitos. Ferramentas que podem ser usadas variam desde sistemas especializados em gerenciamento de requisitos até planilhas e sistemas de banco de dados simples.

O gerenciamento de requisitos precisa de apoio automatizado, e as ferramentas de software para esse gerenciamento devem ser escolhidas durante a fase de planejamento. Você precisa de ferramentas de apoio para:

1. *Armazenamento de requisitos*. Os requisitos devem ser mantidos em um repositório de dados gerenciado e seguro, acessível a todos os envolvidos no processo de engenharia de requisitos.
2. *Gerenciamento de mudanças*. O processo de gerenciamento de mudanças (Figura 4.9) é simplificado quando as ferramentas ativas de apoio estão disponíveis.
3. *Gerenciamento de rastreabilidade*. Como discutido anteriormente, as ferramentas de apoio para rastreabilidade permitem descobrir requisitos relacionados. Algumas ferramentas estão disponíveis, as quais usam técnicas de processamento de linguagem natural para ajudar a descobrir as possíveis relações entre os requisitos.

Figura 4.9 Gerenciamento de mudança de requisitos.

Para sistemas de pequeno porte, podem não ser necessárias ferramentas especializadas no gerenciamento de requisitos. O processo de gerenciamento de requisitos pode ser apoiado por recursos disponíveis nos processadores de texto, planilhas e bancos de dados do PC. No entanto, para sistemas maiores, ferramentas de apoio mais especializadas são necessárias. Incluí links para informações sobre as ferramentas de gerenciamento de requisitos nas páginas do livro no site.

4.7.2 Gerenciamento de mudança de requisitos

Após a aprovação do documento de requisitos, o gerenciamento de mudança de requisitos (Figura 4.9) deve ser aplicado a todas as mudanças propostas aos requisitos de um sistema. O gerenciamento de mudanças é essencial, pois é necessário decidir se os benefícios da implementação de novos requisitos justificam os custos de implementação. A vantagem de se usar um processo formal de gerenciamento de mudanças é que todas as propostas de mudanças são tratadas de forma consistente, e as alterações nos documentos de requisitos são feitas de forma controlada.

Existem três estágios principais em um processo de gerenciamento de mudanças:

1. *Análise de problema e especificação de mudanças.* O processo começa com um problema de requisitos identificado ou, às vezes, com uma proposta específica de mudança. Durante esse estágio, analisa-se o problema ou a proposta de mudança a fim de se verificar sua validade. Essa análise é transmitida a quem solicitou a mudança, que pode responder com uma proposta mais específica de mudança de requisitos ou retirar a solicitação.
2. *Análise de mudanças e custos.* O efeito da mudança proposta é avaliado por meio de informações de rastreabilidade e conhecimentos gerais dos requisitos do sistema. O custo de fazer a mudança é estimado em termos de modificações no documento de requisitos e, se apropriado, no projeto e implementação do sistema. Uma vez que essa análise é concluída, decide-se prosseguir ou não com a mudança de requisitos.
3. *Implementação de mudanças.* O documento de requisitos e, quando necessário, o projeto e implementação do sistema são modificados. Você deve organizar o documento de requisitos para poder fazer alterações sem ampla reformulação ou reorganização. Tal como acontece com os programas, a mutabilidade nos documentos é obtida minimizando-se as referências externas e tornando as seções do documento o mais modular possível. Assim, as seções individuais podem ser alteradas e substituídas sem afetar outras partes do documento.

Se um novo requisito precisa ser implementado com urgência, há sempre a tentação de mudar o sistema e, em seguida, retrospectivamente modificar o documento de requisitos. Esse procedimento deve ser evitado, pois quase inevitavelmente faz com que a especificação de requisitos e a implementação do sistema fiquem defasadas. Uma vez que mudanças no sistema foram feitas, é fácil esquecer de incluir essas alterações no documento de requisitos ou acrescentar informações inconsistentes com a implementação no documento de requisitos.

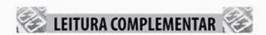
Processos ágeis de desenvolvimento, como Extreme Programming, foram concebidos para lidar com requisitos mutáveis durante o processo de desenvolvimento. Nesses processos, quando um usuário propõe uma mudança nos requisitos, a mudança não passa por um processo formal de gerenciamento de mudanças. Pelo contrário, o usuário tem de priorizar essa mudança e, em caso de alta prioridade, decidir quais recursos do sistema planejados para a próxima iteração devem ser abandonados.

PONTOS IMPORTANTES

- Os requisitos para um sistema de software estabelecem o que o sistema deve fazer e define as restrições sobre seu funcionamento e implementação.
- Os requisitos funcionais são declarações dos serviços que o sistema deve fornecer ou descrições de como alguns processamentos devem ser efetuados.
- Muitas vezes, os requisitos não funcionais restringem o sistema que está sendo desenvolvido e o processo de

desenvolvimento que está sendo usado. Estes podem ser os requisitos de produto, requisitos organizacionais ou requisitos externos. Eles costumam se relacionar com as propriedades emergentes do sistema e, portanto, aplicam-se ao sistema como um todo.

- O documento de requisitos de software é uma declaração acordada dos requisitos do sistema. Deve ser organizado para que ambos — os clientes do sistema e os desenvolvedores de software — possam usá-lo.
- O processo de engenharia de requisitos inclui um estudo da viabilidade, elicitação e análise de requisitos, especificação de requisitos, validação e gerenciamento de requisitos.
- Elicitação e análise de requisitos é um processo iterativo que pode ser representado como uma espiral de atividades — descoberta de requisitos, classificação e organização de requisitos, negociação de requisitos e documentação de requisitos.
- A validação de requisitos é o processo de verificação da validade, consistência, completude, realismo e verificabilidade dos requisitos.
- Mudanças organizacionais, mudanças nos negócios e mudanças técnicas inevitavelmente geram mudanças nos requisitos para um sistema de software. O gerenciamento de requisitos é o processo de gerenciamento e controle dessas mudanças.

LEITURA COMPLEMENTAR

Software Requirements, 2nd edition. Esse livro, projetado para escritores e usuários de requisitos, discute boas práticas de engenharia de requisitos. (WEIGERS, K. M. *Software Requirements*. 2. ed. Microsoft Press., 2003.)

Integrated requirements engineering: A tutorial. Esse é um artigo tutorial que escrevi, no qual discuto as atividades da engenharia de requisitos e como elas podem ser adaptadas para as práticas modernas da engenharia de software. (SOMMERVILLE, I. Integrated requirements engineering: A tutorial. *IEEE Software*, v. 22, n. 1, jan.-fev. 2005.) Disponível em: <http://dx.doi.org/10.1109/MS.2005.13>.

Mastering the Requirements Process, 2nd edition. Um livro bem escrito e fácil de ler, que se baseia em um método específico (VOLERE), mas que também inclui bons conselhos gerais sobre engenharia de requisitos. (ROBERTSON, S.; ROBERTSON, J. *Mastering the Requirements Process*. 2. ed. Addison-Wesley, 2006.)

Research Directions in Requirements Engineering. Esse é um bom levantamento da pesquisa de engenharia de requisitos, que destaca os desafios das futuras pesquisas da área para resolver questões como escala e agilidade. (CHENG, B. H. C.; ATLEE, J. M. Research Directions in Requirements Engineering. *Proc. Conf on Future of Software Engineering*, IEEE Computer Society, 2007.) Disponível em: <http://dx.doi.org/10.1109/FOSE.2007.17>.

EXERCÍCIOS

4.1 Identifique e descreva brevemente os quatro tipos de requisitos que podem ser definidos para um sistema computacional.

4.2 Descubra ambiguidades ou omissões nas seguintes declarações de requisitos para parte de um sistema de emissão de bilhetes:

Um sistema automatizado para emitir bilhetes vende bilhetes de trem. Os usuários selecionam seu destino e inserem um cartão de crédito e um número de identificação pessoal. O bilhete é emitido, e sua conta de cartão de crédito, cobrada. Quando o usuário pressiona o botão de início, é ativado um *display* de menu de destinos possíveis, junto com uma mensagem ao usuário para selecionar um destino. Uma vez que o destino tenha sido selecionado, os usuários são convidados a inserir seu cartão de crédito. Sua validade é verificada e, em seguida, é solicitada ao usuário a entrada de um identificador pessoal. Quando a operação de crédito é validada, o bilhete é emitido.

4.3 Reescreva a descrição anterior usando a abordagem estruturada descrita neste capítulo. Resolva, de um modo apropriado, as ambiguidades identificadas.

4.4 Escreva um conjunto de requisitos não funcionais para o sistema de emissão de bilhetes, definindo sua confiabilidade e tempo de resposta esperados.

4.5 Usando a técnica sugerida neste capítulo, em que as descrições em linguagem natural são apresentadas em formato-padrão, escreva requisitos do usuário plausíveis para as seguintes funções:

- Um sistema de bomba de gasolina autônoma, que inclui um leitor de cartão de crédito. O cliente passa o cartão pelo leitor e, em seguida, especifica a quantidade de combustível requerida. O combustível é liberado, e a conta do cliente, debitada.
- A função de distribuidor de dinheiro em um caixa eletrônico de banco (ATM).
- Os recursos de verificação e correção ortográfica em um editor de texto.

4.6 Sugira como um engenheiro responsável pela elaboração de um sistema de especificação de requisitos pode manter o acompanhamento dos relacionamentos entre requisitos funcionais e não funcionais.

4.7 Usando seu conhecimento de como um caixa eletrônico (ATM) funciona, desenvolva um conjunto de casos de uso que poderia servir de base para o entendimento dos requisitos para um sistema de ATM.

4.8 Quem deve ser envolvido em uma revisão de requisitos? Desenhe um modelo de processo mostrando como uma revisão de requisitos pode ser organizada.

4.9 Quando mudanças emergenciais precisam ser feitas em sistemas, o software do sistema pode precisar ser modificado antes de serem aprovadas as mudanças nos requisitos. Sugira um modelo de um processo para fazer essas modificações de modo a garantir que o documento de requisitos e implementação do sistema não se tornem inconsistentes.

4.10 Você está trabalhando com um usuário de software que contratou seu empregador anterior; juntos, buscam desenvolver um sistema para ele. Você descobre que a interpretação dos requisitos por sua empresa atual é diferente da interpretação de seu empregador anterior. Discuta o que você deve fazer em tal situação. Você sabe que os custos para seu atual empregador aumentarão se as ambiguidades não forem resolvidas. No entanto, você também tem a responsabilidade da confidencialidade com seu empregador anterior.

REFERÊNCIAS

BECK, K. Embracing Change with Extreme Programming. *IEEE Computer*, v. 32, n. 10, 1999, p. 70-78.

CRABTREE, A. *Designing Collaborative Systems*: A Practical Guide to Ethnography. Londres: Springer-Verlag, 2003.

DAVIS, A. M. *Software Requirements*: *Objects, Functions and States*. Englewood Cliffs, NJ: Prentice Hall, 1993.

IEEE. IEEE Recommended Practice for Software Requirements Specifications. In: *IEEE Software Engineering Standards Collection*. Los Alamitos, Ca.: IEEE Computer Society Press, 1998.

JACOBSON, I.; CHRISTERSON, M.; JONSSON, P.; OVERGAARD, G. *Object-Oriented Software Engineering*. Wokingham: Addison-Wesley, 1993.

KOTONYA, G.; SOMMERVILLE, I. *Requirements Engineering*: Processes and Techniques. Chichester, Reino Unido: John Wiley and Sons, 1998.

LARMAN, C. *Applying UML and Patterns*: *An Introduction to Object-oriented Analysis and Design and the Unified Process*. Englewood Cliff, NJ: Prentice Hall, 2002.

MARTIN, D.; RODDEN, T.; ROUNCEFIELD, M.; SOMMERVILLE, I.; VILLER, S. Finding Patterns in the Fieldwork. *Proc. ECSCW'01*. Bonn: Kluwer, 2001, p. 39-58.

MARTIN, D.; ROUNCEFIELD, M.; SOMMERVILLE, I. Applying patterns of interaction to work (re)design: E-government and planning. *Proc. ACM CHI'2002*, ACM Press, 2002, p. 235-242.

MARTIN, D.; SOMMERVILLE, I. Patterns of interaction: Linking ethnomethodology and design. *ACM Trans. on Computer-Human Interaction*, v. 11, n. 1, 2004, p. 59-89.

ROBERTSON, S.; ROBERTSON, J. *Mastering the Requirements Process*. Harlow, Reino Unido: Addison-Wesley, 1999.

SOMMERVILLE, I.; RODDEN, T.; SAWYER, P.; BENTLEY, R.; TWIDALE, M. Integrating ethnography into the requirements engineering process. *Proc. RE'93*, San Diego CA.: IEEE Computer Society Press, 1993, p. 165-173.

STEVENS, P.; POOLEY, R. *Using UML*: *Software Engineering with Objects and Components*. 2. ed. Harlow, Reino Unido: Addison-Wesley, 2006.

SUCHMAN, L. *Plans and Situated Actions*. Cambridge: Cambridge University Press, 1987.

VILLER, S.; SOMMERVILLE, I. Coherence: An Approach to Representing Ethnographic Analyses in Systems Design. *Human-Computer Interaction*, v. 14, n. 1, 2, 1999, p. 9-41.

_______. Ethnographically informed analysis for software engineers. *Int. J. of Human-Computer Studies*, v. 53, n. 1, 2000, p. 169-196.

CAPÍTULO

1 2 3 4 **5** 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Modelagem de sistemas

Objetivos

O objetivo deste capítulo é apresentar alguns tipos de modelos de sistema que podem ser desenvolvidos como parte dos processos de engenharia de requisitos e de projeto de sistema. Com a leitura deste capítulo você:

- compreenderá como os modelos gráficos podem ser usados para representar sistemas de software;
- compreenderá por que diferentes tipos de modelo são necessários e as perspectivas fundamentais de modelagem de sistema de contexto, interação, estrutura e comportamento;
- terá sido apresentado a alguns dos tipos de diagramas da *Unified Modeling Language* (UML), e como eles podem ser usados na modelagem de sistema;
- estará ciente das ideias subjacentes à engenharia dirigida a modelos, com a qual um sistema é gerado automaticamente a partir de modelos estruturais e comportamentais.

Conteúdo

Modelagem de sistema é o processo de desenvolvimento de modelos abstratos de um sistema, em que cada modelo apresenta uma visão ou perspectiva, diferente do sistema. A modelagem de sistema geralmente representa o sistema com algum tipo de notação gráfica, que, atualmente, quase sempre é baseada em notações de UML (linguagem de modelagem unificada, do inglês *Unified Modeling Language*). No entanto, também é possível desenvolver modelos (matemáticos) formais de um sistema, normalmente como uma especificação detalhada do sistema. Neste capítulo, escrevo sobre a modelagem gráfica usando a UML, e no Capítulo 12, escrevo sobre a modelagem formal.

Os modelos são usados durante o processo de engenharia de requisitos para ajudar a extrair os requisitos do sistema; durante o processo de projeto, são usados para descrever o sistema para os engenheiros que o implementam; e, após isso, são usados para documentar a estrutura e a operação do sistema. Você pode desenvolver modelos do sistema existente e do sistema a ser desenvolvido:

1. Modelos do sistema existente são usados durante a engenharia de requisitos. Eles ajudam a esclarecer o que o sistema existente faz e podem ser usados como ponto de partida para discutir seus pontos fortes e fracos. Levam, então, os requisitos para o novo sistema.
2. Modelos do novo sistema são usados durante a engenharia de requisitos para ajudar a explicar os requisitos propostos para outros *stakeholders* do sistema. Os engenheiros usam esses modelos para discutir propostas de projeto e documentar o sistema para a implementação. Em um processo de engenharia dirigida a modelos, é possível gerar uma implementação completa ou parcial do sistema a partir do modelo de sistema.

O aspecto mais importante de um modelo de sistema é que ele deixa de fora os detalhes. Um modelo é uma abstração do sistema a ser estudado, e não uma representação alternativa dele. Idealmente, uma representação de um sistema deve manter todas as informações sobre a entidade representada. Uma abstração deliberadamente simplifica e seleciona as características mais salientes. Por exemplo, no caso muito improvável de este livro ser publicado por capítulos, em um jornal, a apresentação seria uma abstração dos pontos-chave do livro. Ao ser traduzido do inglês para outro idioma, teremos uma representação alternativa. A intenção do tradutor é manter toda a informação como ela foi apresentada em inglês.

A partir de perspecitvas diferentes, você pode desenvolver diversos modelos para representar o sistema. Por exemplo:

1. Uma perspectiva externa, em que você modela o contexto ou o ambiente do sistema.
2. Uma perspectiva de interação, em que você modela as interações entre um sistema e seu ambiente, ou entre os componentes de um sistema.
3. Uma perspectiva estrutural, em que você modela a organização de um sistema ou a estrutura dos dados processados pelo sistema.
4. Uma perspectiva comportamental, em que você modela o comportamento dinâmico do sistema e como ele reage aos eventos.

Essas perspectivas têm muito em comum com a visão 4 + 1, de Kruchten, da arquitetura do sistema (KRUCHTEN, 1995), na qual ele sugere que você deve documentar a arquitetura e organização de um sistema por perspectivas diferentes. No Capítulo 6, discuto a abordagem 4 + 1.

Neste capítulo, uso diagramas definidos em UML (BOOCH et al., 2005; RUMBAUGH et al., 2004), que se tornou uma linguagem de modelagem padrão para modelagem orientada a objetos. A UML tem muitos tipos de diagramas e, dessa forma, apoia a criação de muitos tipos de diferentes modelos de sistema. No entanto, uma pesquisa em 2007 (ERICKSON e SIAU, 2007) mostrou que a maioria dos usuários de UML acredita que cinco tipos de diagramas podem representar a essência de um sistema:

1. Diagramas de atividades, que mostram as atividades envolvidas em um processo ou no processamento de dados.
2. Diagramas de casos de uso, que mostram as interações entre um sistema e seu ambiente.
3. Diagramas de sequência, que mostram as interações entre os atores e o sistema, e entre os componentes do sistema.
4. Diagramas de classe, que mostram as classes de objeto no sistema e as associações entre elas.
5. Diagramas de estado, que mostram como o sistema reage aos eventos internos e externos.

Como não tenho espaço para discutir todos os tipos de diagramas UML, concentro-me em como esses cinco tipos principais são usados na modelagem de sistema.

Ao desenvolver modelos de sistema, muitas vezes você pode ser flexível no uso da notação gráfica. Você não precisa se ater rigidamente aos detalhes de uma notação. O detalhe e o rigor de um modelo dependem de como você pretende usá-lo. Os modelos gráficos são comumente usados de três formas:

1. Como forma de facilitar a discussão sobre um sistema existente ou proposto.
2. Como forma de documentar um sistema já existente.
3. Como uma descrição detalhada de um sistema, que pode ser usada para gerar uma implementação do sistema.

No primeiro caso, o objetivo do modelo é estimular a discussão entre os engenheiros de software envolvidos no desenvolvimento do sistema. Os modelos podem ser incompletos (contanto que cubram os pontos essenciais da discussão) e podem usar a notação de modelagem informalmente. É assim que os modelos são normalmente usados na chamada 'modelagem ágil' (AMBLER e JEFFRIES, 2002). Quando os modelos são usados como documentação, não precisam ser completos, pois você pode querer desenvolver modelos apenas para algumas partes de um sistema. No entanto, esses modelos precisam ser corretos; eles devem usar a notação de forma correta e apresentar uma descrição precisa do sistema.

No terceiro caso, em que os modelos são usados como parte de um processo de desenvolvimento dirigido a modelos, os modelos de sistema precisam ser completos e corretos. A razão para isso é que eles são usados como uma base para gerar o código-fonte do sistema. Portanto, você precisa ter muito cuidado para não confundir símbolos semelhantes, como setas de ponta e blocos, cujos significados são diferentes.

5.1 Modelos de contexto

Em um estágio inicial da especificação de um sistema, você deve decidir os limites do sistema. Isso envolve trabalhar com os *stakeholders* do sistema para decidir qual funcionalidade deve ser incluída no sistema e o que é fornecido pelo ambiente do sistema. Você pode decidir que o apoio automatizado para alguns processos de negócio deve ser implementado, mas outros processos devem ser manuais ou apoiados por sistemas diferentes. Em termos de funcionalidade, você deve olhar para as possíveis sobreposições aos sistemas existentes e decidir onde a nova funcionalidade deve ser implementada. Essas decisões devem surgir no início do processo para limitar os custos e o tempo necessário para compreender os requisitos e o projeto do sistema.

Em alguns casos, a fronteira entre um sistema e seu ambiente é relativamente clara. Por exemplo, quando um sistema automatizado está substituindo um sistema já existente, manual ou informatizado, o ambiente do novo sistema é, geralmente, o mesmo do sistema existente. Em outros casos, existe mais flexibilidade, e, durante o processo de engenharia de requisitos, você decide o que constitui a fronteira entre o sistema e seu ambiente.

Por exemplo, digamos que você esteja desenvolvendo a especificação para o sistema de informação de pacientes para a área da saúde mental. Esse sistema destina-se a gerenciar informações sobre os pacientes que procuram clínicas de saúde mental e os tratamentos prescritos. Ao desenvolver a especificação para esse sistema, é preciso decidir se ele deve se concentrar exclusivamente em coletar informações sobre as consultas (usando outros sistemas para coletar informações pessoais sobre os pacientes) ou se deve também coletar informações pessoais dos pacientes. A vantagem de se basear em outros sistemas de informação de pacientes é que você evita a duplicação de dados. A maior desvantagem, entretanto, é que o uso de outros sistemas pode torná-lo mais lento para o acesso a informações. Se esses sistemas não estão disponíveis, então o MHC-PMS não pode ser usado.

A definição do limite do sistema não é livre de juízos de valor. Interesses sociais e organizacionais podem significar que a posição de limite do sistema pode ser determinada por fatores não técnicos. Por exemplo, um limite do sistema pode ser deliberadamente posicionado de modo que o processo de análise possa ocorrer em um site, pode ser escolhido de forma que um gerente particularmente difícil não precise ser consultado, pode ser posicionado de modo que o custo do sistema seja maior e a equipe de desenvolvimento do sistema necessite expandir-se para projetar e implementar o sistema.

Depois de tomadas algumas decisões a respeito dos limites do sistema, parte da atividade de análise é a definição desse contexto e das dependências que o sistema tem em seu ambiente. Normalmente, a produção de um modelo de arquitetura simples é o primeiro passo para essa atividade.

A Figura 5.1 é um modelo de contexto simples que mostra o sistema de informação de pacientes e os outros sistemas em seu ambiente. A partir da Figura 5.1, você pode ver que o MHC-PMS é conectado a um sistema mais geral de agendamentos e a um sistema de registro de pacientes com o qual compartilha os dados. O sistema também está conectado a sistemas para gerenciamento de relatórios e de alocação de leitos hospitalares e a um

Figura 5.1 O contexto do MHC-PMS

sistema de estatísticas que coleta informações para a pesquisa. Por fim, faz uso de um sistema de prescrições para gerar receitas para a medicação dos pacientes.

Geralmente, os modelos de contexto mostram que o ambiente inclui vários outros sistemas automatizados. No entanto, eles não mostram os tipos de relacionamentos entre os sistemas no ambiente e o sistema que está sendo especificado. Os sistemas externos podem produzir dados para o sistema ou consumir dados deste. Eles podem compartilhar dados com o sistema, podem ser conectados diretamente, por meio de uma rede, ou não ser conectados a coisa alguma. Podem estar fisicamente no mesmo local ou localizados em prédios separados. Todas essas relações podem afetar os requisitos e o projeto do sistema a ser definido e devem ser levadas em conta.

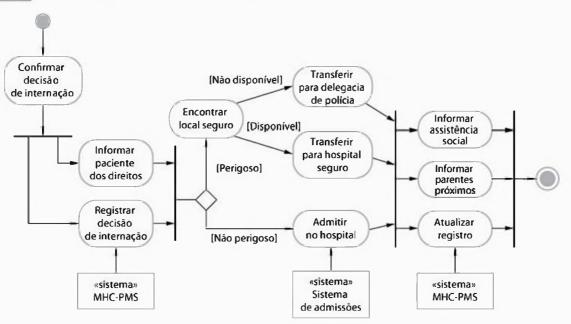
Portanto, os modelos de contexto simples são usados com outros modelos, como modelos de processos de negócio. Estes descrevem os processos humanos e automatizados em que os sistemas de software específicos são usados.

A Figura 5.2 é um modelo de um importante processo de sistema que mostra os processos em que o MHC-PMS é usado. Às vezes, os pacientes que sofrem de problemas de saúde mental podem ser um perigo para os outros e/ou para eles mesmos. Portanto, pode acontecer de serem internados contra a vontade, em um hospital, para receberem tratamento. Essa internação é sujeita a severas salvaguardas jurídicas, por exemplo, a decisão de internar um paciente deve ser regularmente revista para que as pessoas não sejam internadas indefinidamente, sem uma boa razão. Uma das funções do MHC-PMS é garantir que essas salvaguardas sejam implementadas.

A Figura 5.2 é um diagrama de atividades da UML. Os diagramas de atividades são destinados a mostrar as atividades que compõem um processo de sistema e o fluxo de controle de uma atividade para a outra. O início de um processo é indicado por um círculo preenchido; o fim, por um círculo preenchido dentro de outro círculo. Os retângulos com cantos arredondados representam atividades, ou seja, os subprocessos específicos que devem ser realizados. Você pode incluir objetos em diagramas de atividades. Na Figura 5.2, mostrei os sistemas usados para apoiar processos diferentes. Indiquei que esses são sistemas separados usando o recurso de estereótipo da UML.

Em um diagrama de atividades da UML, as setas representam o fluxo de trabalho de uma atividade para outra. Uma barra sólida é usada para indicar coordenação de atividades. Quando o fluxo de mais de uma atividade leva a uma barra sólida, todas essas atividades devem ser concluídas antes de o progresso ser possível. Quando o fluxo de uma barra sólida leva a uma série de atividades, elas podem ser executadas em paralelo. Portanto, na Figura 5.2, as atividades de informar a assistência social e os parentes próximos do paciente podem ser concorrentes à atualização do registo de internação.

As setas podem ser anotadas com guardas que indicam a condição de quando o fluxo é tomado. Na Figura 5.2, você pode ver os guardas que mostram os fluxos para pacientes perigosos e não perigosos para a sociedade. Os

Figura 5.2 Modelo de processos de internação involuntária

pacientes perigosos para a sociedade devem ser internados em uma instalação segura. No entanto, os pacientes suicidas e, portanto, perigosos para eles mesmos, podem ser internados em uma enfermaria apropriada de um hospital.

5.2 Modelos de interação

Todos os sistemas envolvem algum tipo de interação. Pode-se ter interação do usuário, que envolve entradas e saídas, interação entre o sistema que está em desenvolvimento e outros sistemas, ou interação entre os componentes do sistema. A modelagem da interação do usuário é importante, pois ajuda a identificar os requisitos do usuário.

O sistema de modelagem da interação do sistema destaca os problemas de comunicação que podem surgir. A modelagem da interação nos ajuda a compreender se a estrutura proposta para o sistema é suscetível de produzir o desempenho e a confiança requerida do sistema. Nesta seção, trato de duas abordagens relacionadas à modelagem da interação:

1. Modelagem de caso de uso, usada principalmente para modelar interações entre um sistema e atores externos (usuários ou outros sistemas).
2. Diagramas de sequência, usados para modelar interações entre os componentes do sistema, embora os agentes externos também possam ser incluídos.

Os modelos de caso de uso e diagramas de sequência apresentam interações em diferentes níveis de detalhamento e, assim, podem ser usados juntos. Os detalhes das interações envolvidas em um caso de uso de alto nível podem ser documentados em um diagrama de sequência. A UML inclui diagramas de comunicação que podem ser usados para modelar as interações. Como eles são uma representação alternativa de diagramas de sequência, eu não os discuto neste momento. De fato, algumas ferramentas podem gerar um diagrama de comunicação a partir de um diagrama de sequência.

5.2.1 Modelagem de caso de uso

A modelagem de caso de uso foi originalmente desenvolvida por Jacobson et al. (1993) na década de 1990, e foi incorporada ao primeiro *release* da UML (RUMBAUGH et al., 1999). Como discutido no Capítulo 4, a modelagem de caso de uso é amplamente usada para apoiar a elicitação de requisitos. Um caso de uso pode ser tomado como um cenário simples que descreve o que o usuário espera de um sistema.

Cada caso de uso representa uma tarefa discreta que envolve a interação externa com um sistema. Em sua forma mais simples, um caso de uso é mostrado como uma elipse, com os atores envolvidos representados por figuras-palito. A Figura 5.3 mostra um caso de uso do MHC-PMS que representa a tarefa de carregar os dados do MHC-PMS para um sistema mais geral de registro de pacientes. Esse sistema mais geral mantém um resumo dos dados do paciente, em vez de manter os dados sobre cada consulta, que são registrados no MHC-PMS.

Observe que existem dois atores nesse caso de uso: o operador que transfere os dados e o sistema de registro de pacientes. A notação da figura-palito foi originalmente desenvolvida para cobrir a interação humana, mas também é usada para representar outros sistemas externos e hardware. Formalmente, os diagramas de caso de uso devem usar linhas sem setas, pois na UML as setas indicam a direção do fluxo de mensagens. Certamente, em um caso de uso, as mensagens seguem nas duas direções. No entanto, as setas na Figura 5.3 são usadas informalmente para indicar que a(o) recepcionista do médico inicia a operação e os dados são transferidos para o sistema de registro de pacientes.

Figura 5.3 Caso de uso de transferência de dados.

Diagramas de casos de uso dão uma visão simples de uma interação. Logo, é necessário fornecer mais detalhes para entender o que está envolvido. Esses detalhes podem ser uma simples descrição textual, uma descrição estruturada em uma tabela ou um diagrama de sequência, como discutido a seguir. Você deve escolher o formato mais adequado, dependendo do caso de uso, e o nível de detalhamento que você acredita ser necessário no modelo. Para mim, o formato-padrão de tabela é o mais útil. A Tabela 5.1 mostra uma descrição tabular do caso de uso 'Transferir dados'.

Como já discutido no Capítulo 4, diagramas compostos de casos de uso mostram situações diferentes. Às vezes, todas as possíveis interações de um sistema são incluídas em um único diagrama composto de casos de uso. No entanto, isso pode não ser possível, conforme o número de casos de uso. Nesses casos, você pode desenvolver vários diagramas, cada um mostrando casos de uso relacionados. Por exemplo, a Figura 5.4 mostra todos os casos de uso no MHC-PMS em que o ator 'recepcionista do médico' está envolvido.

5.2.2 Diagramas de sequência

Os diagramas de sequência em UML são usados, principalmente, para modelar as interações entre os atores e os objetos em um sistema e as interações entre os próprios objetos. A UML tem uma sintaxe rica para diagramas de sequência, que permite a modelagem de vários tipos de interação. Como não tenho espaço para cobrir todas as possibilidades aqui, concentro-me nos fundamentos desse tipo de diagrama.

Como o nome indica, um diagrama de sequência mostra a sequência de interações que ocorrem durante um caso de uso em particular ou em uma instância de caso de uso. A Figura 5.5 é um exemplo de diagrama de sequência que ilustra os conceitos básicos da notação. Esse diagrama modela as interações envolvidas no caso de uso 'Ver informações de pacientes', em que a recepcionista do médico pode ver algumas informações sobre o paciente.

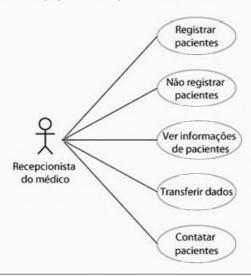
Os objetos e atores envolvidos estão listados na parte superior do diagrama, com uma linha tracejada verticalmente a partir deles. Interações entre objetos são indicadas por setas anotadas. O retângulo na linha tracejada indica a linha da vida do objeto em questão (ou seja, o tempo em que a instância do objeto está envolvida no processamento). Deve-se ler a sequência de interações de cima para baixo. As anotações sobre as setas indicam as chamadas para os objetos, seus parâmetros e os valores de retorno. Nesse exemplo, também mostro a notação para designar as alternativas. Uma caixa nomeada 'alt' é usada com as condições indicadas entre colchetes.

Você pode ler a Figura 5.5 da seguinte maneira:

1. A recepcionista do médico aciona o método VerInfo em uma instância P da classe de objeto InfoPaciente, fornecendo o identificador do paciente (PID, do inglês *patient's identifier*). P é um objeto de interface do usuário, exibido como um formulário que mostra os dados do paciente.
2. A instância P chama o banco de dados para retornar as informações necessárias, fornecendo o identificador da recepcionista, que permite a verificação de proteção (nessa fase, não importa de onde vem esse UID — do inglês, user´s identifier).
3. O banco de dados verifica, com um sistema de autorização, que o usuário está autorizado a essa ação.
4. Se autorizado, as informações de pacientes são retornadas, e um formulário é preenchido na tela do usuário. Se falhar a autorização, aparece uma mensagem de erro.

Tabela 5.1 Descrição tabular do caso de uso 'Transferir dados'

Atores	Recepcionista do médico, sistema de registros de pacientes (PRS, do inglês *patient records system*)
Descrição	Uma recepcionista pode transferir dados do MHC-PMS para um banco de dados geral de registros de pacientes mantido por uma autoridade de saúde. As informações transferidas podem ser atualizadas com as informações pessoais (endereço, telefone etc.) ou com um resumo do diagnóstico e tratamento do paciente.
Dados	Informações pessoais do paciente, resumo do tratamento.
Estímulos	Comando de usuário emitido pela recepcionista do médico.
Resposta	Confirmação de que o PRS foi atualizado.
Comentários	A recepcionista deve ter permissões de proteção adequadas para acessar as informações do paciente e o PRS.

Figura 5.4 Casos de uso envolvendo o papel da 'recepcionista do médico'

A Figura 5.6 é um segundo exemplo de diagrama de sequência do mesmo sistema, que ilustra duas características adicionais. Estas são a comunicação direta entre os atores do sistema e a criação de objetos como parte de uma sequência de operações. Nesse exemplo, um objeto do tipo Sumário é criado para armazenar os dados de resumo que serão enviados para o PRS. Você pode ler esse diagrama da seguinte maneira:

1. A recepcionista inicia sessão (*login*) no PRS.
2. Existem duas opções disponíveis. Elas permitem a transferência direta de informações atualizadas de pacientes para o PRS e a transferência de dados do sumário de saúde do MHC-PMS para o PRS.
3. Em cada caso, as permissões da recepcionista são verificadas por meio do sistema de autorização.
4. As informações pessoais podem ser transferidas diretamente do objeto de interface de usuário para o PRS. Como alternativa, um registro do sumário pode ser criado a partir do banco de dados e, então, o registro é transferido.
5. Após concluir a transferência, o PRS emite uma mensagem de *status* e o usuário fecha a sessão (logoff).

Figura 5.5 Diagrama de sequência para 'Ver informações de pacientes'

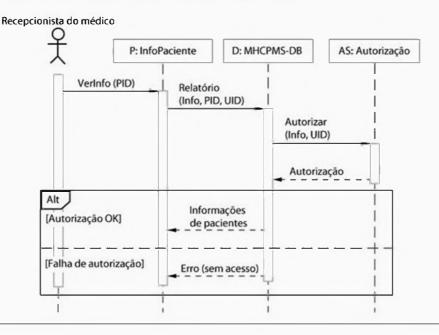
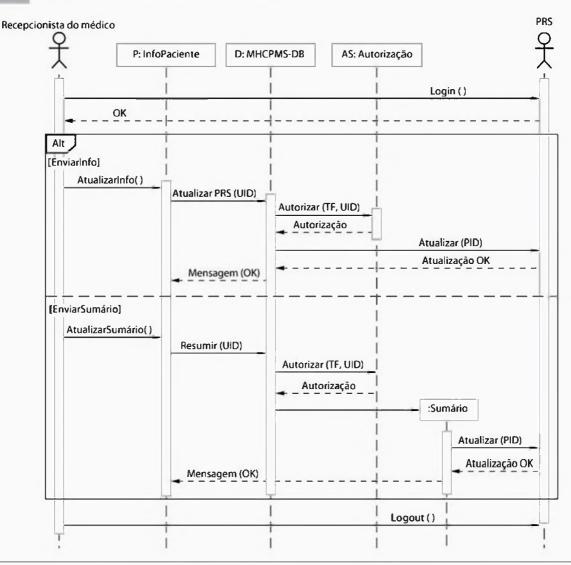



Figura 5.6 Diagrama de sequência para 'Transferir dados'

A menos que você esteja usando diagramas de sequência para geração de código ou documentação detalhada, você não precisa incluir todas as interações nesses diagramas. Se você desenvolver modelos de sistemas no início do processo de desenvolvimento para dar apoio à engenharia de requisitos e projeto de alto nível, acontecerão muitas interações que dependem de decisões de implementação. Por exemplo, na Figura 5.6, a decisão de como obter o identificador de usuário para verificação de autorização pode ser adiada. Em uma implementação, isso pode envolver a interação com um objeto Usuário, mas, nesse momento, essa decisão não é importante e, por isso, não deve ser incluída no diagrama de sequência.

5.3 Modelos estruturais

Os modelos estruturais de softwares exibem a organização de um sistema em termos de seus componentes e seus relacionamentos. Os modelos estruturais podem ser modelos estáticos, que mostram a estrutura do projeto do sistema, ou modelos dinâmicos, que mostram a organização do sistema quando ele está em execução. Essas são duas características distintas — a organização da dinâmica de um sistema, como um conjunto de *threads* (pequenos processos) que interagem entre si pode ser muito diferente de um modelo estático de componentes do sistema.

É possível criar modelos estruturais de um sistema quando se está discutindo e projetando sua arquitetura. O projeto de arquitetura é um tema particularmente importante na engenharia de software, e componentes, pacotes e diagramas de implantação da UML podem ser usados quando da apresentação dos modelos de arquitetura. Nos capítulos 6, 18 e 19, vários aspectos da arquitetura de software e modelagem de arquitetura serão discutidos. Nesta seção, focalizo o uso de diagramas de classe para modelar a estrutura estática das classes de objeto em um sistema de software.

5.3.1 Diagramas de classe

Os diagramas de classe são usados no desenvolvimento de um modelo de sistema orientado a objetos para mostrar as classes de um sistema e as associações entre essas classes. Em poucas palavras, uma classe de objeto pode ser pensada como uma definição geral de um tipo de objeto do sistema. Uma associação é um link entre classes que indica algum relacionamento entre essas classes. Consequentemente, cada classe pode precisar de algum conhecimento sobre sua classe associada.

Quando você está desenvolvendo modelos, durante os estágios iniciais do processo de engenharia de software, os objetos representam algo no mundo real, como um paciente, uma receita médica, um médico etc. Enquanto uma aplicação é desenvolvida, geralmente é necessário definir objetos adicionais de implementação que são usados para fornecer a funcionalidade requerida do sistema. Aqui, vou me concentrar na modelagem de objetos do mundo real, como parte dos requisitos ou processos iniciais de projeto de software.

Os diagramas de classe em UML podem ser expressos em diferentes níveis de detalhamento. Quando você está desenvolvendo um modelo, o primeiro estágio geralmente é o de olhar para o mundo, identificar os objetos essenciais e representá-los como classes. A maneira mais simples de fazer isso é escrever o nome da classe em uma caixa. Você também pode simplesmente observar a existência de uma associação, traçando uma linha entre as classes. Por exemplo, a Figura 5.7 é um diagrama de classes simples, mostrando duas classes — 'Paciente' e 'Registro de paciente' — com uma associação entre elas.

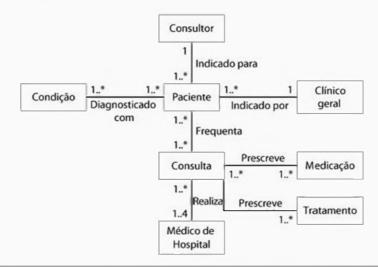
Na Figura 5.7, vemos outra característica dos diagramas de classe: a capacidade de mostrar quantos objetos estão envolvidos na associação. Nesse exemplo, cada extremidade da associação é anotada com um 1, o que significa que existe um relacionamento 1:1 entre objetos dessas classes. Ou seja, cada paciente tem exatamente um registro, e cada registro mantém informações sobre um mesmo paciente. Como você poderá ver, mais tarde, em exemplos, outras multiplicidades são possíveis. Você pode definir o número exato de objetos envolvidos ou, usando um *, conforme a Figura 5.8, indicar que há um número indefinido de objetos envolvidos na associação.

A Figura 5.8 desenvolve esse tipo de diagrama de classe para mostrar que os objetos da classe Paciente também estão envolvidos em relacionamentos com uma série de outras classes. Nesse exemplo, vou mostrar que é possível nomear as associações para dar ao leitor uma indicação do tipo de relacionamento que pode existir. A UML também permite a especificação do papel dos objetos participantes na associação.

Nesse nível de detalhamento, os diagramas de classe são parecidos com modelos semânticos de dados. Modelos semânticos de dados são usados no projeto de banco de dados. Eles mostram as entidades dos dados, seus atributos associados e as relações entre essas entidades. Essa abordagem de modelagem foi proposta pela primeira vez em meados da década de 1970, por Chen (1976); desde então, diversas variantes têm sido desenvolvidas (CODD, 1979; HAMMER e McLEOD, 1981; HULL e KING, 1987), todas com a mesma forma básica.

A UML não inclui uma notação específica para essa modelagem de banco de dados, pois supõe um processo de desenvolvimento orientado a objetos e modela dados usando objetos e seus relacionamentos. No entanto, pode-se usar a UML para representar um modelo semântico de dados. Sob um modelo semântico de dados, podemos pensar em entidades como classes de objetos simplificados (não possuem operações), em atributos como atributos da classe de objetos e em relações como as associações nomeadas entre classes de objetos.

Figura 5.7 Classes e associação em UML

Figura 5.8 Classes e associações no MHC-PMS

Ao mostrar as associações entre classes, é conveniente representar essas classes da maneira mais simples possível. Para defini-las com mais detalhes, você adiciona informações sobre seus atributos (características de um objeto) e operações (as coisas que você pode pedir a um objeto). Por exemplo, um objeto Paciente terá o atributo Endereço, e você pode incluir uma operação chamada MudarEndereço, que é chamada quando um paciente indica ter mudado de endereço. Na UML, você mostra os atributos e operações alargando o retângulo simples que representa uma classe. Isso é ilustrado na Figura 5.9, em que:

1. O nome da classe de objeto está na seção superior.
2. Os atributos de classe estão na seção do meio. O que deve incluir os nomes dos atributos e, opcionalmente, seus tipos.
3. As operações (chamadas métodos, em Java e em outras linguagens de programação OO) associadas à classe de objeto estão na seção inferior do retângulo.

A Figura 5.9 mostra os possíveis atributos e operações da classe Consulta. Nesse exemplo, assumo que os médicos gravam anotações de voz para registrar os detalhes da consulta, transcritas posteriormente. Para prescrever a medicação, o médico envolvido deve usar o método Prescrever e gerar uma prescrição eletrônica.

Figura 5.9 A classe de consultas

Consulta
Médicos Data Horário Clinica Motivo Medicação prescrita Tratamento prescrito Anotações de voz Transcrições ...
Novo ( ) Prescrever ( ) RegistrarAnotações ( ) Transcrever ( ) ...

5.3.2 Generalização

A generalização é uma técnica que usamos todos os dias para gerenciar a complexidade. Ao invés de aprender as características detalhadas de cada entidade que nós experimentamos, colocamos essas entidades em classes mais gerais (animais, carros, casas etc.) e aprendemos suas características. Isso nos permite inferir que os diferentes membros dessas classes têm algumas características em comum (por exemplo, os esquilos e os ratos são roedores). Podemos fazer afirmações genéricas que se aplicam a todos os membros da classe (por exemplo, todos os roedores têm dentes para roer).

Na modelagem de sistemas, muitas vezes é útil examinar as classes de um sistema para ver se há espaço para a generalização. Isso significa que as informações comuns serão mantidas em um único lugar. Essa é uma boa prática de projeto, pois quer dizer que, se mudanças são propostas, você não tem de olhar para todas as classes no sistema para ver se são afetadas pela mudança. Em linguagens orientadas a objeto como Java, a generalização é implementada usando os mecanismos de herança de classe construídos dentro da linguagem.

A UML tem um tipo específico de associação para denotar a generalização, como mostra a Figura 5.10. A generalização é representada como uma seta apontando para a classe mais geral. Isso mostra que 'Clínicos gerais' e 'Médicos de hospital' podem ser generalizados como 'Médicos', e que existem três tipos de 'Médicos de hospital' — aqueles que acabaram de se formar em medicina e precisam ser supervisionados ('Médico residente'); aqueles que podem trabalhar sem supervisão, como parte de uma equipe médica ('Médico qualificado'); e 'Consultores', que são os médicos mais experientes com a responsabilidade total pela decisão tomada.

Em uma generalização, os atributos e operações associados com as classes de nível alto também estão associados com as de nível baixo. Em essência, as classes de nível baixo são subclasses que herdam os atributos e as operações de suas superclasses. Essas classes de nível mais baixo, em seguida, adicionam mais atributos e operações específicas. Por exemplo, todos os médicos têm um nome e um número de telefone, todos os médicos de hospital têm um número de equipe e um departamento; mas não os clínicos gerais — esses trabalham de forma independente, e por isso não têm esses atributos. No entanto, eles têm um nome e endereço para prática. Essa situação é ilustrada na Figura 5.11, que mostra parte da hierarquia de generalização que eu estendi com atributos da classe. As operações associadas com a classe Médico destinam-se a registrar e remover o registro desse médico com o MHC-PMS.

5.3.3 Agregação

No mundo real, objetos são frequentemente compostos de diferentes partes. Por exemplo, um pacote de estudo para um curso pode ser composto de um livro, *slides* do PowerPoint, *quizzes* e recomendações para leitura posterior. Às vezes, em um modelo de sistema, você precisa ilustrar isso. A UML fornece um tipo especial de associação entre as classes chamada agregação, que significa que um objeto (o todo) é composto de outros objetos (as

Figura 5.10 Uma hierarquia de generalização

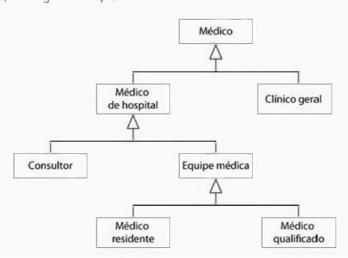
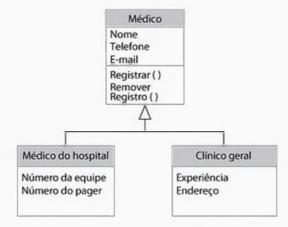



Figura 5.11 Uma hierarquia de generalização com detalhes adicionais

partes). Para mostrar isso, usamos uma forma de losango ao lado da classe que representa o todo. Isso é mostrado na Figura 5.12, que mostra que um registro de um paciente é uma composição de um Paciente e uma ou mais Consultas.

5.4 Modelos comportamentais

Modelos comportamentais são modelos do comportamento dinâmico do sistema quando está em execução. Eles mostram o que acontece ou deve acontecer quando o sistema responde a um estímulo de seu ambiente. Você pode pensar nesse estímulo como sendo de dois tipos:

1. *Dados* — alguns dados que chegam precisam ser processados pelo sistema.
2. *Eventos* — alguns eventos que acontecem disparam o processamento do sistema. Eles podem ter dados associados, mas nem sempre esse é o caso.

Muitos sistemas de negócios são sistemas de processamento de dados essencialmente dirigidos por dados. Eles são controlados pela entrada de dados no sistema com processamento relativamente baixo de eventos externos. Seu processamento envolve uma sequência de ações nos dados e a geração de uma saída. Por exemplo, um sistema de cobrança de telefonia aceitará as informações sobre as chamadas feitas por um cliente, calculará os custos dessas chamadas e gerará uma conta para ser enviada ao cliente. Em contrapartida, sistemas de tempo real são muitas vezes dirigidos por eventos com mínimo processamento de dados. Por exemplo, um sistema de comutação de telefonia fixa responde a eventos como 'receptor fora do gancho' gerando um tom de discagem, ou o pressionamento de teclas de um telefone capturando o número do telefone etc.

Figura 5.12 A associação por agregação

5.4.1 Modelagem orientada a dados

Modelos dirigidos a dados mostram a sequência de ações envolvidas no processamento de dados de entrada e a geração de uma saída associada. Eles são particularmente úteis durante a análise de requisitos, pois podem ser usados para mostrar, do início ao fim, o processamento de um sistema. Ou seja, eles mostram toda a sequência de ações, desde uma entrada sendo processada até a saída correspondente, que é a resposta do sistema.

Modelos dirigidos a dados estavam entre os primeiros modelos gráficos de software. Na década de 1970, os métodos estruturados, como Análise Estruturada de DeMarco (DeMARCO, 1978), apresentaram os diagramas de fluxo de dados (DFDs, do inglês *data-flow diagrams*) como forma de ilustrar as etapas de processamento em um sistema. Modelos de fluxo de dados são úteis porque analisar e documentar como os dados associados a um determinado processo se movem pelo sistema ajuda os analistas e projetistas a entenderem o que está acontecendo. Diagramas de fluxo de dados são simples e intuitivos, e normalmente é possível explicá-los aos potenciais usuários do sistema, que, então, podem participar na validação do modelo.

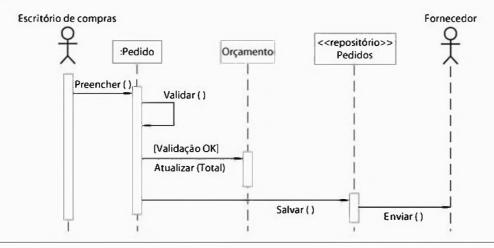
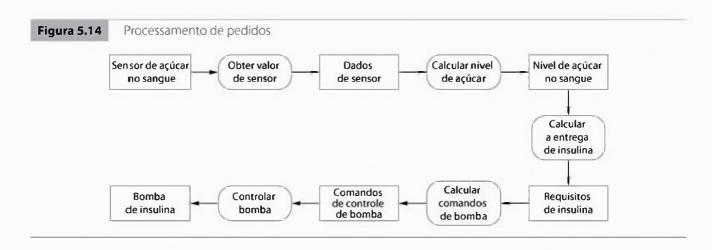
A UML não oferece apoio a diagramas de fluxo de dados, pois estes foram inicialmente propostos e usados para modelagem de processamento de dados. A razão para isso é que os DFDs se centram sobre as funções do sistema e não reconhecem os objetos do sistema. No entanto, devido aos sistemas dirigidos a dados serem tão comuns no mundo dos negócios, a UML 2.0 introduziu diagramas de atividades, semelhantes a diagramas de fluxo de dados. Por exemplo, a Figura 5.13 mostra a cadeia de processamento envolvida no software da bomba de insulina. Nesse diagrama, você pode ver as etapas de processamento (representadas como atividades) e os dados fluindo entre essas etapas (representadas como objetos).

Uma forma alternativa de mostrar a sequência de processamento em um sistema é o uso de diagramas de sequência da UML. Já vimos como esses diagramas podem ser usados para modelar interações, mas, se você desenhar de modo que as mensagens sejam enviadas apenas da esquerda para a direita, então verá que elas mostram o processamento de dados sequencial no sistema. A Figura 5.14 ilustra isso, usando um modelo de sequência de processamento de um pedido e enviando-o para um fornecedor. Modelos de sequência destacam os objetos em um sistema, enquanto os diagramas de fluxo de dados destacam as funções. O diagrama de fluxo de dados equivalente para o processamento do pedido é apresentado nas páginas do livro na Internet.

5.4.2 Modelagem dirigida a eventos

Modelagem dirigida a eventos mostra como o sistema reage a eventos externos e internos. Ela é baseada na suposição de que um sistema tem um número finito de estados e que os eventos (estímulos) podem causar uma transição de um estado para outro. Por exemplo, um sistema de controle de uma válvula pode mover-se de um estado 'Válvula aberta' para um estado 'Válvula fechada', quando um comando do operador (o estímulo) é recebido. Essa percepção de um sistema é particularmente adequada para sistemas de tempo real. A modelagem baseada em eventos foi introduzida em métodos de projeto de tempo real, como as propostas por Ward e Mellor (1985) e Harel (1987, 1988).

Figura 5.13 Modelo de atividades de funcionamento da bomba de insulina

Figura 5.14 Processamento de pedidos

A UML apoia a modelagem baseada em eventos com diagramas de estado, baseados em *Statecharts* (HAREL, 1987, 1988). Os diagramas de estado mostram os estados do sistema e os eventos que causam transições de um estado para outro. Eles não mostram o fluxo de dados dentro do sistema, mas podem incluir informações adicionais sobre os processamentos realizados em cada estado.

Para ilustrar a modelgem dirigida a eventos, uso um exemplo de software de controle para um forno de micro-ondas muito simples. Fornos de micro-ondas reais são muito mais complexos que esse sistema, mas o sistema simplificado é mais fácil de entender. Esse micro-ondas simples tem um interruptor para selecionar potência total ou meia, um teclado numérico para inserir o tempo de cozimento, um botão iniciar/cancelar e um *display* alfanumérico.

Vamos assumir que a sequência de ações no uso do micro-ondas seja:

1. Selecionar o nível de potência (ou meia potência ou potência total).
2. Introduzir o tempo de cozimento usando um teclado numérico.
3. Pressionar 'Iniciar', e os alimentos são cozidos para o tempo selecionado.

Por razões de segurança, o forno não deve operar quando a porta está aberta e, no fim do cozimento, uma campainha é acionada. O forno tem um *display* alfanumérico muito simples, que é usado para mostrar vários alertas e mensagens de advertência.

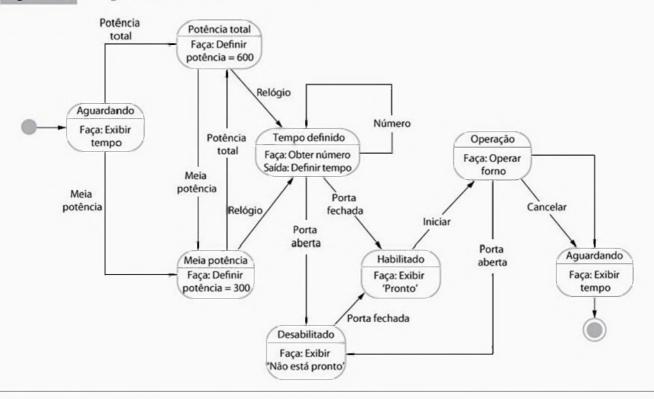
Em diagramas de estado da UML, retângulos arredondados representam os estados do sistema. Eles podem incluir uma breve descrição (após 'Faça') das ações tomadas nesse estado. As setas rotuladas representam estímulos que forçam uma transição de um estado para outro. Você pode indicar os estados inicial e final usando círculos preenchidos, como em um diagrama de atividades.

A partir da Figura 5.15, você pode ver que o sistema começa em um estado de espera e responde inicialmente ao botão de potência total ou ao de meia potência. Após selecionar um dos botões, os usuários podem mudar de ideia e pressionar o outro botão. O tempo é definido, e, se a porta estiver fechada, o botão 'Iniciar' é habilitado. Pressionando-se esse botão, começa a operação do forno, e o cozimento ocorrerá para o tempo especificado. Esse é o fim do ciclo de cozimento, e o sistema retorna ao estado de espera.

A notação UML permite-lhe indicar a atividade que ocorre em um estado. Em uma especificação detalhada do sistema, você precisa fornecer mais detalhes tanto sobre os estímulos como sobre os estados do sistema. Vê-se isso na Tabela 5.2, que mostra uma descrição tabular de cada estado, e como são gerados os estímulos que forçam as transições de estado.

O problema com a modelagem baseada em estados é que o número de possíveis estados aumenta rapidamente. Para modelos de sistemas de grande porte, portanto, você precisa esconder detalhes nos modelos. Uma maneira de fazer isso é usando a noção de um superestado que encapsula um número de estados distintos. Esse superestado, em um modelo de alto nível, parece um único estado, mas depois, em um diagrama separado, é ampliado para mostrar mais detalhes. Para ilustrar esse conceito, considere o estado 'Operação' na Figura 5.15. Esse é um superestado que pode ser expandido, conforme ilustrado na Figura 5.16.

O estado 'Operação' inclui uma série de subestados, o que mostra que a operação inicia com uma verificação de *status* e, se qualquer problema é descoberto, um alarme é indicado, e a operação, desabilitada. O cozimento envolve a execução do gerador de micro-ondas para o tempo especificado; na conclusão, uma campainha é emitida. Se a porta for aberta durante a operação, o sistema se move para o estado desabilitado, como mostra a Figura 5.15.

Figura 5.15 Diagrama de estados de um forno de micro-ondas

5.5 Engenharia dirigida a modelos

A engenharia dirigida a modelos (MDE, do inglês *model-based engineering*) é uma abordagem do desenvolvimento de software segundo a qual os modelos, em vez de programas, são as saídas principais do processo de desenvolvimento (KENT, 2002; SCHMIDT, 2006). Os programas executados em um hardware/software são, então, gerados automaticamente a partir dos modelos. Os defensores da MDE argumentam que esta aumenta o nível de abstração na engenharia de software, e, dessa forma, os engenheiros não precisam mais se preocupar com detalhes da linguagem de programação ou com as especificidades das plataformas de execução.

A engenharia dirigida a modelos tem suas raízes na arquitetura dirigida a modelos (MDA, do inglês *model-driven architecture*), proposta pelo Object Management Group (OMG), em 2001, como um novo paradigma de desenvolvimento de software. A engenharia e a arquitetura dirigidas a modelos são, frequentemente, vistas como a mesma coisa. No entanto, penso que a MDE tem um alcance maior que a MDA. Como discuto mais adiante nesta seção, a MDA concentra-se nos estágios de projeto e implementação do processo de desenvolvimento de software, ao passo que a MDE se preocupa com todos os aspectos do processo de engenharia de software. Assim, temas como a engenharia de requisitos baseada em modelos, processos de software para o desenvolvimento baseado em modelos e testes baseados em modelos fazem parte, atualmente, da MDE, mas não da MDA.

Embora a MDA esteja em uso desde 2001, a engenharia baseada em modelo está ainda em um estágio inicial de desenvolvimento e não está claro se ela terá ou não um efeito significativo sobre as práticas da engenharia de software. Os principais argumentos a favor e contra a MDE são:

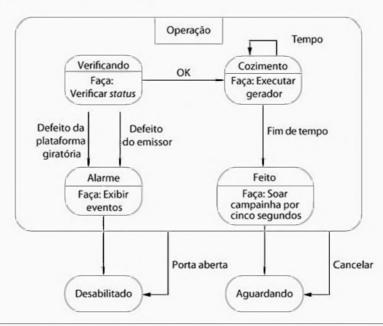
1. *A favor da MDE.* A engenharia dirigida a modelos permite que os engenheiros pensem em sistemas em alto nível de abstração, sem preocupação com detalhes de implementação. Isso reduz a probabilidade de erros, acelera o processo de projeto e implementação e permite a criação de modelos de aplicação reusáveis, independentes de plataforma. Por meio de ferramentas poderosas, as implementações do sistema podem ser geradas para diferentes plataformas a partir do mesmo modelo. Portanto, para adaptar o sistema a uma nova tecnologia de plataforma, é necessário apenas escrever um tradutor para essa plataforma. Quando este estiver disponível, todos os modelos independentes de plataforma podem ser rapidamente reimplantados na nova plataforma.

Tabela 5.2 Estados e estímulos para o forno de micro-ondas

Estado	**Descrição**
Aguardando	O forno está aguardando uma entrada. O *display* mostra a hora atual.
Meia potência	A potência do forno é definida para 300 watts. O *display* mostra 'Meia potência'.
Potência total	A potência do forno é definida para 600 watts. O *display* mostra 'Potência total'.
Tempo definido	O tempo de cozimento é definido como valor de entrada do usuário. O *display* mostra o tempo de cozimento selecionado e é atualizado conforme o tempo definido.
Desabilitado	A operação do forno está desabilitada por questões de segurança. A iluminação interna do forno está acesa. O *display* mostra 'Não está pronto'.
Habilitado	A operação do forno está habilitada. A iluminação interna do forno está desligada. O *display* mostra 'Pronto'.
Operação	Forno em operação. A iluminação interna está acesa. O *display* mostra a contagem regressiva do relógio. No fim do cozimento, a campainha soa por cinco segundos. A luz do forno está acesa. O *display* mostra 'Cozimento completo', enquanto a campainha está soando.

Estímulos	**Descrição**
Meia potência	O usuário pressionou o botão de meia potência.
Potência total	O usuário pressionou o botão de potência total.
Relógio	O usuário pressionou um dos botões do relógio.
Número	O usuário pressionou uma tecla numérica.
Porta aberta	O interruptor da porta do forno não está fechado.
Porta fechada	O interruptor da porta do forno está fechado.
Iniciar	O usuário pressionou o botão Iniciar.
Cancelar	O usuário pressionou o botão Cancelar.

2. *Contra a MDE.* Como dito anteriormente, os modelos são uma boa maneira de facilitar as discussões sobre um projeto de software. No entanto, as abstrações apoiadas pelo modelo nem sempre são corretas para a implementação. Assim, você pode criar modelos informais de projeto, mas depois, ao implementar o sistema, usar um pacote configurável de prateleira. Além disso, os argumentos para a independência da plataforma são válidos apenas para sistemas de grande porte e duradouros, em que as plataformas se tornam obsoletas ao longo do tempo de vida do sistema. De qualquer forma, para essa classe de sistemas, sabemos que a implementação não é o grande problema; engenharia de requisitos, proteção e confiança, integração com sistemas legados e testes são mais significativos.

Figura 5.16 Operação do forno de micro-ondas

Existem significativas histórias de sucesso da MDE relatadas pelo OMG em suas páginas na Internet <www.omg.org/mda/products_success.htm>, e a abordagem é usada em grandes empresas, como a IBM e a Siemens. As técnicas têm sido utilizadas com sucesso no desenvolvimento de grandes sistemas de software com ciclos duradouros, como sistemas de gerenciamento do tráfego aéreo. No entanto, no momento em que este livro foi escrito, as abordagens dirigidas a modelos não estavam sendo amplamente usadas pela engenharia de software. Como os métodos formais de engenharia de software, que discuto no Capítulo 12, eu acredito que a MDE seja uma evolução importante. No entanto, como é o caso com os métodos formais, não está claro se os custos e riscos das abordagens dirigidas a modelos ultrapassam os possíveis benefícios.

5.5.1 Arquitetura dirigida a modelos

Arquitetura dirigida a modelos (KLEPPE et al., 2003; MELLOR et al., 2004;. STAHL e VOELTER, 2006) é uma abordagem para projeto e implementação de software centrada em modelos, que usa um subconjunto de modelos da UML para descrever um sistema. Nesta abordagem, são criados modelos em diferentes níveis de abstração. De um modelo independente de plataforma em nível alto é possível, em princípio, gerar um programa de trabalho sem intervenção manual.

O método MDA recomenda a produção de três tipos de modelos abstratos de sistema:

1. Um modelo de computação independente (CIM, do inglês *computation independent model*), que modela as importantes abstrações do domínio usado no sistema. Às vezes, os CIMs são chamados "modelos de domínio". Você pode desenvolver vários CIMs diferentes, refletindo diferentes visões do sistema. Por exemplo, pode-se ter um CIM de proteção, em que se identificam importantes abstrações de proteção, tais como CIM de um ativo, um papel e um registro de paciente, no qual se descrevem abstrações tais como pacientes, consultas etc.
2. Um modelo independente de plataforma (PIM, do inglês *platform independent model*), que modela a operação do sistema sem referência a sua implementação. O PIM geralmente é descrito por meio de modelos da UML que mostram a estrutura estática do sistema e como ele responde a eventos externos e internos.
3. Modelos específicos de plataforma (PSM, do inglês *platform specific models*) são as transformações do modelo independente de plataforma com um PSM separado para cada plataforma de aplicação. Em princípio, pode haver camadas de PSM, com cada uma acrescentando alguns detalhes específicos. Assim, o PSM de primeiro nível pode ser um *middleware* específico, mas independente do banco de dados. Quando um banco de dados específico for escolhido, um PMS específico de dados pode ser gerado.

Como eu já disse, as transformações entre esses modelos podem ser definidas e aplicadas automaticamente por ferramentas de software. Essa situação é ilustrada na Figura 5.17, que também mostra um nível final de transformação automática. Uma transformação é aplicada ao PMS para gerar o código executável que roda na plataforma de software designada.

No momento em que este livro foi escrito, a tradução automática CIM para PIM ainda estava em estágio de protótipo de pesquisa. É pouco provável que, em um futuro próximo, ferramentas de tradução completamente automáticas estejam disponíveis. Em um futuro previsível, a intervenção humana, indicada por uma figura palito na Figura 5.15, ainda será necessária. Os CIMs estão relacionados, e parte do processo de tradução pode envolver ligar conceitos em diferentes CIMs. Por exemplo, o conceito de um papel em um CIM de proteção pode ser mapeado para o conceito de uma equipe em um CIM de hospital. Mellor e Balcer (2002) dão o nome de'pontes'à informação que apoia o mapeamento de um CIM para outro.

A tradução de PIMs em PSMs está mais madura, e existem várias ferramentas comerciais disponíveis que fornecem tradutores de PIMs para plataformas comuns como Java e J2EE. Elas contam com uma extensa biblioteca de regras e padrões específicos de plataforma para conversão de PIM para PSM. Pode haver vários PSMs para cada PIM no sistema. Se um sistema de software é planejado para ser executado em diferentes plataformas (por exemplo, J2EE e .NET), é necessário apenas manter o PIM. Os PSMs para cada plataforma são gerados automaticamente. Essa situação é ilustrada na Figura 5.18.

Embora as ferramentas de apoio MDA incluam tradutores específicos de plataforma, são frequentes os casos em que só oferecem apoio parcial para a tradução de PIMs em PSMs. Na maioria dos casos, o ambiente de execução de um sistema é mais do que a plataforma de execução padrão (por exemplo, J2EE, .NET etc.). Também inclui outros sistemas aplicativos, bibliotecas de aplicativos específicas de uma empresa e bibliotecas de interface de usuário. Uma vez que estes variam significativamente de uma empresa para outra, não está disponível um apoio de ferramentas padrão. Portanto, quando a MDA é introduzida, pode ser necessário criar tradutores para fins especiais, para que sejam consideradas as características do ambiente local. Em alguns casos (por exemplo, para a geração da interface de usuário), pode ser impossível a tradução PIM para PSM totalmente automatizada.

Existe uma relação desconfortável entre os métodos ágeis e a MDA. A noção de modelagem inicial extensiva contradiz as ideias fundamentais do manifesto ágil, e suspeito que poucos desenvolvedores ágeis se sintam confortáveis com a MDE. Os desenvolvedores da MDA afirmam que ela é destinada a apoiar uma abordagem iterativa para o desenvolvimento, e por isso pode ser usada dentro de métodos ágeis (MELLOR et al., 2004). Se as transformações podem ser totalmente automatizadas e um programa completo pode ser gerado a partir de um PIM, então, em princípio, a MDA pode ser usada em um processo ágil de desenvolvimento, pois nenhuma codificação separada seria necessária. No entanto, pelo que sei, não existem ferramentas de MDA que deem apoio às práticas como testes de regressão e desenvolvimento dirigido a testes.

5.5.2 UML executável

A ideia fundamental por trás da MDE é que a transformação completamente automatizada de modelos para códigos deve ser possível. Para conseguir isso, é preciso ser capaz de construir modelos gráficos com a semântica bem-definida. Você também precisa encontrar uma maneira de agregar informações aos modelos gráficos sobre

Figura 5.17 Transformações de MDA

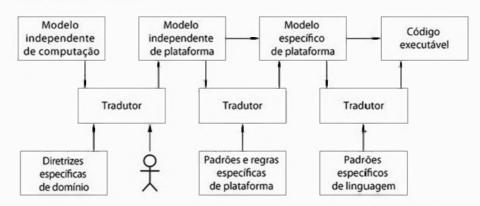
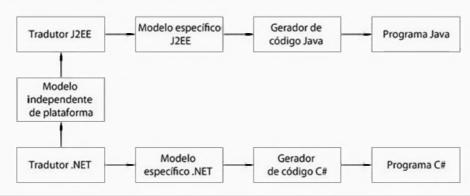



Figura 5.18 Vários modelos específicos de plataforma

as maneiras como as operações definidas no modelo são implementadas. Isso é possível usando-se um subconjunto da UML 2, chamada UML executável ou xUML (MELLOR e BALCER, 2002). Eu não tenho espaço para descrever os detalhes da xUML, assim, apresento um breve resumo de suas principais características.

A UML foi concebida como uma linguagem de apoio e documentação de projetos de software, e não como uma linguagem de programação. Os projetistas da UML não estavam preocupados com os detalhes semânticos da linguagem, mas com sua expressividade. Eles introduziram noções úteis, como diagramas de caso de uso, que ajudam com o projeto, mas são demasiadamente informais para apoiar a execução. Para criar um subconjunto executável da UML, o número de tipos de modelo, portanto, foi drasticamente reduzido para três tipos de modelos-chave:

1. Modelos de domínio, que identificam os principais interesses no sistema. Estes são definidos por meio de diagramas de classe da UML que incluem objetos, atributos e associações.
2. Modelos de classe, em que as classes são definidas, junto com seus atributos e operações.
3. Modelos de estado, em que um diagrama de estado está associado a cada classe e é usado para descrever o ciclo de vida da classe.

O comportamento dinâmico do sistema pode ser especificado declarativamente usando-se a linguagem de restrição de objetos (OCL, do inglês *object constraint language*), ou pode ser expresso em linguagem de ação da UML. A linguagem de ação é como uma linguagem de programação de nível muito alto, na qual você pode se referir a objetos e seus atributos e especificar ações a serem realizadas.

PONTOS IMPORTANTES

- Um modelo é uma visão abstrata de um sistema que ignora alguns detalhes do sistema. Modelos de sistema complementares podem ser desenvolvidos para mostrar o contexto, as interações, a estrutura e o comportamento do sistema.
- Modelos de contexto mostram como um sistema que está sendo modelado é posicionado em um ambiente com outros sistemas e processos. Eles ajudam a definir os limites do sistema a ser desenvolvido.
- Diagramas de caso de uso e diagramas de sequência são usados para descrever as interações entre o usuário do sistema que será projetado e usuários ou outros sistemas. Os casos de uso descrevem as interações entre um sistema e atores externos. Diagramas de sequência acrescentam mais informações a eles, mostrando as interações entre os objetos do sistema.
- Os modelos estruturais mostram a organização e a arquitetura de um sistema. Os diagramas de classe são usados para definir a estrutura estática de classes em um sistema e suas associações.
- Os modelos comportamentais são usados para descrever o comportamento dinâmico de um sistema em execução. Podem ser modelados a partir da perspectiva dos dados processados pelo sistema ou pelos eventos que estimulam respostas de um sistema.
- Os diagramas de atividades podem ser usados para modelar o processamento de dados, em que cada atividade representa uma etapa do processo.

- Os diagramas de estado são usados para modelar o comportamento de um sistema em resposta a eventos internos ou externos.
- A engenharia dirigida a modelos é uma abordagem de desenvolvimento de software em que um sistema é representado como um conjunto de modelos que pode ser automaticamente transformado em código executável.

LEITURA COMPLEMENTAR

Requirements Analysis and System Design. Esse livro concentra-se na análise de sistemas de informação e discute como diferentes modelos da UML podem ser usados no processo de análise. (MACIASZEK, L. *Requirements Analysis and System Design*. Addison-Wesley, 2001.)

MDA Distilled: Principles of Model-driven Architecture. Essa é uma introdução concisa e acessível para o método MDA. Foi escrita por entusiastas, de modo que o livro diz muito pouco sobre possíveis problemas com essa abordagem. (MELLOR, S. J.; SCOTT, K.; WEISE, D. *MDA Distilled: Principles of Model-driven Architecture*. Addison-Wesley, 2004.)

Using UML: Software Engineering with Objects and Components, 2nd ed. Uma introdução curta e legível para o uso da UML na especificação e projeto de sistemas. Esse livro é excelente para aprender e entender a UML, embora não seja uma descrição completa da notação. (STEVENS, P.; POOLEY, R. *Using UML: Software Engineering with Objects and Components*. 2. ed. Addison-Wesley, 2006.)

EXERCÍCIOS

5.1 Explique por que é importante modelar o contexto de um sistema que está sendo desenvolvido. Dê dois exemplos de possíveis erros que podem ocorrer, caso os engenheiros de software não entendam o contexto do sistema.

5.2 Como você poderia usar um modelo de um sistema que já existe? Explique por que nem sempre é necessário que um modelo de sistema seja completo e correto. O mesmo seria verdadeiro caso você estivesse desenvolvendo um modelo de um novo sistema?

5.3 Você foi convidado para desenvolver um sistema que vai ajudar no planejamento de grandes eventos e festas, como casamentos, festas de formatura, festas de aniversário etc. Usando um diagrama de atividades, modele o contexto do processo para um sistema que mostre as atividades envolvidas no planejamento de uma festa (reserva de um local, organização dos convites etc.) e os elementos do sistema que podem ser usados em cada etapa.

5.4 Para o MHC-PMS, proponha um conjunto de casos de uso que ilustre as interações entre um médico que atende pacientes e prescreve medicamentos e tratamentos e o MHC-PMS.

5.5 Desenvolva um diagrama de sequência que mostre as interações envolvidas quando um estudante se registra para um curso em uma universidade. Os cursos podem ter a inscrição limitada, então o processo de registro deve incluir verificações de vagas disponíveis. Suponha que o estudante acesse um catálogo de cursos eletrônicos para saber mais sobre os cursos disponíveis.

5.6 Olhe atentamente como as mensagens e caixas de correio são representadas no sistema de e-mail que você usa. Modele as classes de objetos que poderiam ser usadas na implementação do sistema para representar uma caixa postal e uma mensagem de correio eletrônico.

5.7 Baseado em sua experiência com um caixa eletrônico, desenhe um diagrama de atividades que modele o processamento de dados envolvido quando um cliente retira dinheiro da máquina.

5.8 Desenhe um diagrama de sequência para o mesmo sistema. Explique por que você pode querer desenvolver ambos, diagramas de atividades e de sequência, ao modelar o comportamento de um sistema.

5.9 Desenhe diagramas de estado do software de controle para:

- Uma máquina de lavar automática, com programas diferentes para diferentes tipos de roupas.
- O software para um DVD player.
- Um sistema de atendimento telefônico que grava as mensagens recebidas e exibe o número de mensagens aceitas em um LED (*Light Emitting Diode*). O sistema deve permitir ao cliente do telefone discar a

partir de qualquer localização, digitar uma sequência de números (identificados como tons) e ouvir todas as mensagens gravadas.

5.10 Você é um gerente de engenharia de software e sua equipe propõe que a engenharia dirigida a modelos deve ser usada para desenvolver um novo sistema. Que fatores você deve levar em conta ao decidir se deve ou não introduzir essa nova abordagem ao desenvolvimento de software?

REFERÊNCIAS

AMBLER, S. W.; JEFFRIES, R. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Nova York: John Wiley & Sons, 2002.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *The Unified Modeling Language User Guide*. 2. ed. Boston: Addison-Wesley, 2005.

CHEN, P. The entity relationship model — Towards a unified view of data. *ACM Trans. on Database Systems*, v. 1, n. 1, 1976, p. 9-36.

CODD, E. F. Extending the database relational model to capture more meaning. *ACM Trans. on Database Systems*, v. 4, n. 4, 1979, p. 397-434.

DeMARCO, T. *Structured Analysis and System Specification*. Nova York: Yourdon Press, 1978.

ERICKSON, J.; SIAU, K. Theoretical and practical complexity of modeling methods. *Comm. ACM*, v. 50, n. 8, 2007, p. 46-51.

HAMMER, M.; McLEOD, D. Database descriptions with SDM: A semantic database model. *ACM Trans. on Database Sys.*, v. 6, n. 3, 1981, p. 351-386.

HAREL, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, v. 8, n. 3, 1987, p. 231-274.

______. On visual formalisms. *Comm. ACM*, v. 31, n. 5, 1988, p. 514-530.

HULL, R.; KING, R. Semantic database modeling: Survey, applications and research issues. *ACM Computing Surveys*, v. 19, n. 3, 1987, p. 201-260.

JACOBSON, I.; CHRISTERSON, M.; JONSSON, P.; OVERGAARD, G. *Object-Oriented Software Engineering*. Wokingham: Addison-Wesley, 1993.

KENT, S. Model-driven engineering. *Proc. 3rd Int. Conf. on Integrated Formal Methods*, 2002, p. 286-298.

KLEPPE, A.; WARMER, J.; BAST, W. *MDA Explained: The Model Driven Architecture — Practice and Promise*. Boston: Addison-Wesley, 2003.

KRUCHTEN, P. The 4 + 1 view model of architecture. *IEEE Software*, v. 11, n. 6, 1995, p. 42-50.

MELLOR, S. J.; BALCER, M. J. *Executable UML*. Boston: Addison-Wesley, 2002.

MELLOR, S. J.; SCOTT, K.; WEISE, D. *MDA Distilled: Principles of Model-driven Architecture*. Boston: Addison-Wesley, 2004.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified Modeling Language Reference Manual*. Reading, Mass.: Addison--Wesley, 1999.

______. *The Unified Modeling Language Reference Manual*. 2. ed. Boston: Addison-Wesley, 2004.

SCHMIDT, D. C. Model-Driven Engineering. *IEEE Computer*, v. 39, n. 2, 2006, p. 25-31.

STAHL, T.; VOELTER, M. *Model-Driven Software Development: Technology, Engineering, Management*. Nova York: John Wiley & Sons, 2006.

WARD, P.; MELLOR, S. *Structured Development for Real-time Systems*. Englewood Cliffs, NJ: Prentice Hall, 1985.

CAPÍTULO

1 2 3 4 5 **6** 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Projeto de arquitetura

Objetivos

O objetivo deste capítulo é introduzir os conceitos da arquitetura de software e do projeto de arquitetura. Depois da leitura, você:

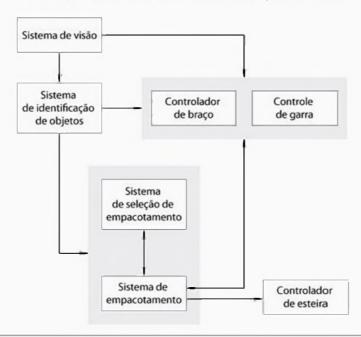
- compreenderá por que o projeto de arquitetura de software é importante;
- compreenderá as decisões necessárias sobre a arquitetura de sistema durante o processo de projeto de arquitetura;
- terá sido apresentado aos padrões de arquitetura, bem como às maneiras já experimentadas de organizar as arquiteturas de sistema, que podem ser reusadas em projetos de sistemas;
- conhecerá os padrões de arquiteturas que muitas vezes são usados em diferentes tipos de sistemas de aplicações, incluindo sistemas de processamento de transações e os sistemas de processamento de linguagens.

Conteúdo

O projeto de arquitetura está preocupado com a compreensão de como um sistema deve ser organizado e com a estrutura geral desse sistema. No modelo do processo de desenvolvimento de software, o projeto de arquitetura é o primeiro estágio no processo de projeto de software, como mostra o Capítulo 2. É o elo crítico entre o projeto e a engenharia de requisitos, pois identifica os principais componentes estruturais de um sistema e os relacionamentos entre eles. O resultado do processo de projeto de arquitetura é um modelo de arquitetura que descreve como o sistema está organizado em um conjunto de componentes de comunicação.

Em processos ágeis, geralmente se aceita que um estágio inicial do processo de desenvolvimento se preocupe com o estabelecimento de uma arquitetura global do sistema. O desenvolvimento incremental de arquiteturas geralmente não é bem-sucedido. Embora a refatoração de compontentes em resposta às mudanças costume ser relativamente fácil, a refatoração de uma arquitetura é geralmente cara.

Para ajudar na compreensão do que quero dizer com arquitetura de sistema, considere a Figura 6.1. Ela mostra um modelo abstrato da arquitetura de um sistema de controle robotizado de empacotamento, que mostra os componentes que precisam ser desenvolvidos. Esse sistema robotizado pode embalar diferentes tipos de objeto. Ele usa um componente de visão para selecionar objetos em uma esteira, identificar o tipo de objeto e selecionar o tipo correto de empacotamento. Em seguida, o sistema retira os objetos da esteira de entrega para serem empacotados. Então, o sistema coloca os objetos empacotados em outra esteira. O modelo de arquitetura mostra esses componentes e os relacionamentos entre eles.

Figura 6.1 A arquitetura de um sistema de controle robotizado de empacotamento

Na prática, existe uma considerável sobreposição entre os processos de engenharia de requisitos e de projeto de arquitetura. Idealmente, uma especificação de sistema não deve incluir todas as informações do projeto. Mas essa não é a realidade, exceto para sistemas muito pequenos. A decomposição de arquitetura é normalmente necessária para estruturar e organizar a especificação. Portanto, como parte do processo de engenharia de requisitos, você poderá propor uma arquitetura abstrata de sistema em que seja possível associar grupos de funções ou recursos do sistema aos componentes em larga escala ou subsistemas. Você pode, então, usar essa decomposição para discutir com os *stakeholders* os requisitos e recursos do sistema.

Você pode projetar as arquiteturas de software em dois níveis de abstração, o que eu chamo de *arquiteturas em pequena e grande escala*:

1. A arquitetura em pequena escala está preocupada com a arquitetura de programas individuais. Nesse nível, estamos preocupados com a maneira como um programa individual é decomposto em componentes. Este capítulo está mais preocupado com as arquiteturas de programas.
2. A arquitetura em grande escala preocupa-se com a arquitetura de sistemas corporativos complexos que incluem outros sistemas, programas e componentes de programas. Esses sistemas empresariais estão distribuídos por diversos computadores, que podem pertencer e ser geridos por diferentes empresas. Eu trato da arquitetura em grande escala nos capítulos 18 e 19, nos quais discuto os sistemas de arquiteturas distribuídas.

A arquitetura de software é importante, pois afeta o desempenho e a robustez, bem como a capacidade de distribuição e de manutenibilidade de um sistema (BOSCH, 2000). Como discute Bosch, os componentes individuais implementam os requisitos funcionais do sistema. Os requisitos não funcionais dependem da arquitetura do sistema — a forma como esses componentes estão organizados e se comunicam. Em muitos sistemas, os requisitos não funcionais são também influenciados por componentes individuais, mas não há dúvida de que a arquitetura de sistema é a influência dominante.

Bass et al. (2003) discutem três vantagens de projetar e documentar, explicitamente, a arquitetura de software:

1. *Comunicação de* stakeholders. A arquitetura é uma apresentação de alto nível do sistema e pode ser usada como um foco de discussão por uma série de diferentes *stakeholders*.
2. *Análise de sistema.* Tornar explícita a arquitetura do sistema, em um estágio inicial de seu desenvolvimento, requer alguma análise. As decisões de projeto de arquitetura têm um efeito profundo sobre a possibilidade de o sistema atender ou não aos requisitos críticos, como desempenho, confiabilidade e manutenibilidade.
3. *Reúso em larga escala.* Um modelo de uma arquitetura de sistema é uma descrição compacta e gerenciável de como um sistema é organizado e como os componentes interoperam. A arquitetura do sistema geralmente é a mesma para sistemas com requisitos semelhantes e, por isso, pode apoiar o reúso de software em grande escala.

Como explico no Capítulo 16, talvez seja possível desenvolver arquiteturas de linha de produto nas quais a mesma arquitetura é reusada em todo um conjunto de sistemas relacionados.

Hofmeister et al. (2000) propõem uma arquitetura de software que pode servir, em primeiro lugar, como um plano de projeto para a negociação de requisitos de sistema, e, em segundo lugar, como um meio de estruturar as discussões com os clientes, desenvolvedores e gerentes. Eles também sugerem que seja uma ferramenta essencial para o gerenciamento da complexidade, pois esconde detalhes e permite que os projetistas se centrem nas abstrações-chave do sistema.

Em geral, as arquiteturas de sistema são modeladas por meio de diagramas de blocos simples, como na Figura 6.1. No diagrama, cada caixa representa um componente. Caixas dentro de caixas indicam que o componente foi decomposto em subcomponentes. As setas significam que os dados e/ou sinais de controle são passados de um componente a outro na direção das setas. Você pode ver muitos exemplos desse tipo de modelo de arquitetura no catálogo de arquitetura de software de Booch (BOOCH, 2009).

Os diagramas de blocos apresentam uma imagem de alto nível da estrutura do sistema, de forma que as pessoas de diferentes disciplinas, envolvidas no processo de desenvolvimento do sistema, possam compreender facilmente. No entanto, apesar de seu amplo uso, Bass et al. (2003) não gostam de diagramas de blocos informais para descrever uma arquitetura. Eles alegam que esses diagramas informais são pobres representações de arquitetura, que não mostram o tipo de relacionamentos entre os componentes do sistema nem as propriedades dos componentes visíveis externamente.

A aparente contradição entre a prática e a teoria de arquitetura surge porque existem duas maneiras de usar um modelo de arquitetura de um programa:

1. *Para facilitar a discussão sobre o projeto do sistema.* Uma visão de arquitetura de um sistema de alto nível é útil para a comunicação com os *stakeholders* do sistema e para o planejamento do projeto, pois não é rica em detalhes. Os *stakeholders* podem relacionar e entender uma visão abstrata do sistema e, então, discutir o sistema como um todo, sem se confundirem com detalhes. O modelo de arquitetura identifica os principais componentes que devem ser desenvolvidos para que os gerentes possam começar a designar pessoas para planejarem o desenvolvimento desses sistemas.
2. *Como forma de documentar uma arquitetura que foi projetada.* O objetivo é produzir um modelo completo de sistema que mostre seus diferentes componentes, suas interfaces e conexões. O argumento para isso é que essa descrição detalhada da arquitetura facilita a compreensão e o desenvolvimento do sistema.

Diagramas de bloco são uma forma adequada de, durante o processo de projeto, descrever a arquitetura do sistema; também são uma boa maneira de apoiar a comunicação entre as pessoas envolvidas no processo. Em muitos projetos, eles são a única documentação de arquitetura que existe. No entanto, se a arquitetura de um sistema deve ser bem documentada, é melhor usar uma notação com semântica bem definida para a descrição de arquitetura. Contudo, como discuto na Seção 6.2, algumas pessoas pensam que a documentação detalhada não é útil e não vale o custo de seu desenvolvimento.

6.1 Decisões de projeto de arquitetura

O projeto de arquitetura é um processo criativo no qual você projeta uma organização de sistema para satisfazer aos requisitos funcionais e não funcionais de um sistema. Por ser um processo criativo, as atividades no âmbito do processo dependem do tipo de sistema a ser desenvolvido, a formação e experiência do arquiteto de sistema e os requisitos específicos para o sistema. Por isso, é útil pensar em projeto de arquitetura como uma série de decisões, em vez de uma sequência de atividades.

Durante o processo de projeto de arquitetura, os arquitetos de sistema precisam tomar uma série de decisões estruturais que afetam profundamente o sistema e seu processo de desenvolvimento. Com base em seus conhecimentos e experiência, eles precisam considerar as seguintes questões fundamentais sobre o sistema:

1. Existe uma arquitetura genérica de aplicação que pode atuar como um modelo para o sistema que está sendo projetado?
2. Como o sistema será distribuído por meio de um número de núcleos ou processadores?
3. Que padrões ou estilos de arquitetura podem ser usados?
4. Qual será a abordagem fundamental para se estruturar o sistema?
5. Como os componentes estruturais do sistema serão decompostos em subcomponentes?
6. Que estratégia será usada para controlar o funcionamento dos componentes do sistema?

7. Qual a melhor organização de arquitetura para satisfazer os requisitos não funcionais do sistema?
8. Como o projeto de arquitetura será avaliado?
9. Como a arquitetura do sistema deve ser documentada?

Embora cada sistema de software seja único, sistemas no mesmo domínio de aplicação frequentemente têm arquiteturas similares, que refletem os conceitos fundamentais desse domínio. Por exemplo, linhas de produto de aplicação são as aplicações construídas em torno de uma arquitetura central com variantes que satisfazem aos requisitos específicos do cliente. Ao projetar uma arquitetura de sistema, você precisa decidir o que seu sistema e as classes de uma aplicação mais gerais têm em comum, e quanto conhecimento dessas arquiteturas de aplicação você pode reusar. Discuto arquiteturas genéricas de aplicação na Seção 6.4, e linhas de produtos de aplicação no Capítulo 16.

Para sistemas embutidos e sistemas projetados para computadores pessoais, geralmente existe apenas um processador, e você não terá de projetar uma arquitetura distribuída para o sistema. No entanto, a maioria dos sistemas de grande porte são atualmente sistemas distribuídos em que o software é distribuído em vários computadores diferentes. A escolha da arquitetura de distribuição é uma decisão importante que afeta o desempenho e a confiabilidade do sistema. Esse é um tema importante em si mesmo e no Capítulo 18 eu o discuto separadamente.

A arquitetura de um sistema de software pode se basear em um determinado padrão ou estilo de arquitetura. Um padrão de arquitetura é uma descrição de uma organização do sistema (GARLAN e SHAW, 1993), como uma organização cliente-servidor ou uma arquitetura em camadas. Os padrões de arquitetura capturam a essência de uma arquitetura que tem sido usada em diferentes sistemas de software. Ao tomar decisões sobre a arquitetura de um sistema, você deve conhecer os padrões comuns, bem como saber onde eles podem ser usados e quais são seus pontos fortes e fracos. Na Seção 6.3, discutiremos uma série de padrões frequentemente usados.

A ideia de Garlan e Shaw, de um estilo de arquitetura (estilo e padrão passaram a significar a mesma coisa), abrange as questões 4 a 6 da lista anteriormente apresentada. Você precisa escolher a estrutura mais adequada, como cliente-servidor ou em camadas, que permitirá o cumprimento dos requisitos do sistema. Para decompor unidades estruturais do sistema, você precisa decidir sobre a estratégia para a decomposição de componentes em subcomponentes. As abordagens que você pode usar permitem a implementação de diferentes tipos de arquitetura. Finalmente, no processo de modelagem de controle, você toma decisões sobre como a execução de componentes é controlada. Você desenvolve um modelo geral dos relacionamentos de controle entre as diversas partes do sistema.

Devido à estreita relação entre os requisitos não funcionais e a arquitetura do software, o estilo e a estrutura da arquitetura particular que você escolhe para um sistema devem depender dos requisitos não funcionais do sistema:

1. *Desempenho*. Se o desempenho for um requisito crítico, a arquitetura deve ser projetada para localizar as operações críticas dentro de um pequeno número de componentes, com todos esses componentes implantados no mesmo computador, em vez de distribuídos pela rede. Isso pode significar o uso de alguns componentes relativamente grandes, em vez de pequenos de baixa granularidade, que reduzem o número de comunicações entre eles. Você também pode considerar a organização do sistema de *run-time*, que permite que o sistema seja replicado e executado em diferentes processadores.
2. *Proteção*. Se a proteção for um requisito crítico, deve ser usada uma estrutura em camadas para a arquitetura, com os ativos mais críticos protegidos nas camadas mais internas, com alto nível de validação de proteção aplicado a essas camadas.
3. *Segurança*. Se a segurança for um requisito crítico, a arquitetura deve ser concebida de modo que as operações relacionadas com a segurança estejam localizadas em um único componente ou em um pequeno número de componentes. Isso reduz os custos e os problemas de validação de segurança e torna possível fornecer sistemas de proteção relacionados que podem desligar o sistema de maneira segura em caso de falha.
4. *Disponibilidade*. Se a disponibilidade for um requisito crítico, a arquitetura deve ser projetada para incluir componentes redundantes, de modo que seja possível substituir e atualizar componentes sem parar o sistema. No Capítulo 13 eu descrevo a arquitetura de dois sistemas que toleram defeitos, para sistemas de alta disponibilidade.
5. *Manutenção*. Se a manutenção for um requisito crítico, a arquitetura do sistema deve ser projetada a partir de compontentes autocontidos de baixa granularidade que podem ser rapidamente alterados. Os produtores de dados devem ser separados dos consumidores, e as estruturas de dados compartilhados devem ser evitadas.

Obviamente, há um conflito potencial entre algumas dessas arquiteturas. Por exemplo, o uso de componentes de grande porte melhora o desempenho, e o uso de componentes pequenos, de baixa granularidade, além de melhorar a manutenibilidade. Se o desempenho e a manutenibilidade forem requisitos importantes do sistema, então alguns compromissos devem ser encontrados. Em alguns casos, isso pode ser conseguido por meio de diferentes padrões ou estilos de arquitetura para diferentes partes do sistema.

Avaliar um projeto de arquitetura é difícil, pois o verdadeiro teste de uma arquitetura é quão bem o sistema satisfaz aos requisitos funcionais e não funcionais quando em uso. No entanto, você pode fazer alguma avaliação, comparando seu projeto contra arquiteturas de referência ou padrões genéricos de arquitetura. A descrição de Bosch (2000) das características não funcionais dos padrões de arquitetura também pode ajudar na avaliação de arquiteturas.

6.2 Visões da arquitetura

Na introdução deste capítulo, expliquei que os modelos de arquitetura de um sistema de software podem ser usados para focar a discussão sobre os requisitos de software ou de projeto. Como alternativa, podem ser usados para documentar um projeto para que este possa ser usado como base para um projeto e uma implementação mais detalhados e para a futura evolução do sistema. Nesta seção, discuto duas questões relevantes para essas duas opções:

1. Que visões ou perspectivas são úteis ao se projetar e documentar uma arquitetura de sistema?
2. Quais notações devem ser usadas para se descrever modelos de arquitetura?

É impossível representar todas as informações relevantes sobre a arquitetura de um sistema em um único modelo de arquitetura, pois cada modelo mostra apenas uma visão ou perspectiva do sistema. Pode mostrar como um sistema é decomposto em módulos, como os processos de *run-time* interagem, ou as diferentes formas como são distribuídos os componentes do sistema através de uma rede. Tudo isso é útil em momentos diferentes; portanto, para ambos, projeto e documentação, geralmente você precisa apresentar múltiplas visões da arquitetura de software.

Existem opiniões diferentes sobre para que as visões são necessárias. Kruchten (1995), em seu bem conhecido modelo de visão 4 + 1 de arquitetura de software, sugere que deve haver quatro visões fundamentais de arquitetura, relacionadas usando-se casos de uso ou cenários. As visões que ele sugere são:

1. A visão lógica, que mostra as abstrações fundamentais do sistema como objetos ou classes de objetos. Nessa visão, deveria ser possível relacionar os requisitos de sistema com as entidades.
2. A visão de processo, que mostra como, no tempo de execução, o sistema é composto de processos interativos. Essa visão é útil para fazer julgamentos sobre as características não funcionais do sistema, como desempenho e disponibilidade.
3. A visão de desenvolvimento, que mostra como o software é decomposto para o desenvolvimento, ou seja, apresenta a distribuição do software em componentes que são implementados por um único desenvolvedor ou por uma equipe de desenvolvimento. Essa visão é útil para gerentes de software e programadores.
4. Uma visão física, que mostra o hardware do sistema e como os componentes de software são distribuídos entre os processadores. Essa visão é útil para os engenheiros de sistemas que estão planejando uma implantação do sistema.

Hofmeister et al. (2000) sugerem o uso de visões semelhantes, mas acrescentam a noção de uma visão conceitual. Essa é uma visão abstrata do sistema que pode ser a base para a decomposição de requisitos de alto nível em especificações mais detalhadas, ajuda os engenheiros a tomarem decisões sobre os componentes que podem ser reusados e representa uma linha de produtos (assunto discutido no Capítulo 16) ao invés de um único sistema. A Figura 6.1, que descreve a arquitetura de um robô de empacotamento, é um exemplo de uma visão conceitual do sistema.

Na prática, as visões conceituais são, quase sempre, desenvolvidas durante o processo de projeto e são usadas para apoiar a tomada de decisões de arquitetura. Elas são uma maneira de comunicar a essência de um sistema para os diferentes *stakeholders*. Durante o processo de projeto, quando diferentes aspectos do sistema são discutidos, outras visões também podem ser desenvolvidas, mas não há necessidade de uma descrição completa de todas as perspectivas. Também pode ser possível associar os padrões de arquitetura, discutidos na próxima seção, com as diferentes visões de um sistema.

Existem opiniões divergentes a respeito do uso da UML para descrições de arquitetura pelos arquitetos de software (CLEMENTS et al., 2002). Uma pesquisa de 2006 (LANGE et al., 2006) mostrou que, quando a UML foi usada, na maioria das vezes foi aplicada de forma solta e informal. Os autores desse trabalho argumentam que esse é um procedimento negativo. Eu não concordo com essa opinião. A UML foi projetada para descrever sistemas orientados a objetos e, no estágio do projeto de arquitetura, muitas vezes você quer descrever os sistemas em um nível maior de abstração. Classes de objetos estão muito próximas da implementação para serem úteis na descrição de arquitetura.

Eu não acho a UML útil durante o processo de projeto; prefiro as notações informais, mais rápidas de se escrever e que podem ser facilmente desenhadas em um quadro branco. A UML tem mais valor quando você está documentando uma arquitetura em detalhes ou usando o desenvolvimento dirigido a modelos, conforme discutido no Capítulo 5.

Vários pesquisadores têm proposto o uso de linguagens de descrição de arquitetura (ADLs, do inglês *Architectural Description Languages*) mais especializadas (BASS et al., 2003) para descrever as arquiteturas de sistema. Os elementos básicos de ADLs são componentes e conectores, que incluem regras e diretrizes para arquiteturas bem formadas. No entanto, devido a sua natureza especializada, especialistas de domínio e aplicação consideram as ADLs difíceis de entender e usar, o que torna difícil avaliar sua utilidade para engenharia de software prática. As ADLs projetadas para um determinado domínio (por exemplo, sistemas de automóveis) podem ser usadas como base para o desenvolvimento dirigido a modelos. No entanto, acredito que os modelos e as notações informais, como a UML, continuarão a ser a forma mais comumente usada para documentar arquiteturas de sistema.

Os usuários de métodos ágeis alegam que, na maior parte do tempo, a documentação detalhada de projeto não é usada. E, portanto, desenvolvê-la seria um desperdício de tempo e dinheiro. Pessoalmente, eu concordo com essa opinião e penso que, na maioria dos sistemas, não vale a pena desenvolver uma descrição de arquitetura detalhada a partir dessas quatro visões. Você deve desenvolver visões úteis para a comunicação, e não se preocupar se sua documentação de arquitetura está completa ou não. No entanto, a exceção é quando você está desenvolvendo sistemas críticos, quando você precisa fazer uma análise detalhada da confiança do sistema. Pode ser necessário convencer os reguladores externos de que o sistema está de acordo com suas regras, e, então, a documentação completa da arquitetura pode ser necessária.

6.3 Padrões de arquitetura

A ideia de padrões como uma forma de apresentar, compartilhar e reusar o conhecimento sobre sistemas de software é hoje amplamente usada. O gatilho para isso foi a publicação de um livro sobre os padrões de projeto orientado a objetos (GAMMA et al., 1995), o que levou ao desenvolvimento de outros tipos de padrões, como padrões para o projeto organizacional (COPLIEN e HARRISON, 2004), padrões de usabilidade (USABILITY GROUP, 1998), interação (MARTIN e SOMMERVILLE, 2004), gerenciamento de configuração (BERCZUK e APPLETON, 2002), e assim por diante. Os padrões de arquitetura foram propostos na década de 1990 sob o nome de 'estilos de arquitetura' (SHAW e GARLAN, 1996), com uma série de cinco volumes de manuais sobre a arquitetura de software orientada a padrões, publicados entre 1996 e 2007 (BUSCHMANN et al., 1996; BUSCHMANN et al., 2007a; BUSCHMANN et al., 2007b; KIRCHERE e JAIN, 2004; SCHMIDT et al., 2000).

Nesta seção, apresento os padrões de arquitetura e descrevo brevemente alguns padrões comumente usados em diferentes tipos de sistemas. Para obter mais informações a respeito dos padrões e seu uso, você deve consultar manuais de padrões publicados.

Você pode pensar em um padrão de arquitetura como uma descrição abstrata, estilizada, de boas práticas experimentadas e testadas em diferentes sistemas e ambientes. Assim, um padrão de arquitetura deve descrever uma organização de sistema bem-sucedida em sistemas anteriores. Deve incluir informações de quando o uso desse padrão é adequado, e seus pontos fortes e fracos.

Por exemplo, a Tabela 6.1 descreve o conhecido padrão MVC (modelo-visão-controlador, do inglês *Model-View-Controller*). Esse padrão é a base do gerenciamento de interação em muitos sistemas baseados em Web. A descrição estilizada de padrão inclui o nome do padrão, uma breve descrição (com um modelo gráfico associado), e um exemplo do tipo de sistema em que o padrão é usado (novamente, talvez com um modelo gráfico). Você também deve incluir informações sobre quando o padrão deve ser usado e suas vantagens e desvantagens. Modelos gráficos da arquitetura associados com o padrão MVC são mostrados nas figuras 6.2 e 6.3. Estes modelos apresentam a arquite-

Tabela 6.1 O padrão modelo-visão-controlador (MVC)

Nome	**MVC (Modelo-Visão-Controlador)**
Descrição	Separa a apresentação e a interação dos dados do sistema. O sistema é estruturado em três componentes lógicos que interagem entre si. O componente Modelo gerencia o sistema de dados e as operações associadas a esses dados. O componente Visão define e gerencia como os dados são apresentados ao usuário. O componente Controlador gerencia a interação do usuário (por exemplo, teclas, cliques do mouse etc.) e passa essas interações para a Visão e o Modelo. Veja a Figura 6.2.
Exemplo	A Figura 6.3 mostra a arquitetura de um sistema aplicativo baseado na Internet, organizado pelo uso do padrão MVC.
Quando é usado	É usado quando existem várias maneiras de se visualizar e interagir com dados. Também quando são desconhecidos os futuros requisitos de interação e apresentação de dados.
Vantagens	Permite que os dados sejam alterados de forma independente de sua representação, e vice-versa. Apoia a apresentação dos mesmos dados de maneiras diferentes, com as alterações feitas em uma representação aparecendo em todas elas.
Desvantagens	Quando o modelo de dados e as interações são simples, pode envolver código adicional e complexidade de código.

tura a partir de diferentes visões — a Figura 6.2 é uma visão conceitual, e a Figura 6.3 mostra uma possível arquitetura *run-time*, quando esse padrão é usado para gerenciamento de interações em um sistema baseado em Web.

Em uma pequena seção de um capítulo geral, é impossível descrever todos os padrões genéricos que podem ser usados no desenvolvimento de software. Em vez disso, apresento alguns exemplos de padrões amplamente usados e que capturam os bons princípios de projeto de arquitetura. No site do livro, há mais exemplos de padrões genéricos de arquitetura.

6.3.1 Arquitetura em camadas

As noções de separação e independência são fundamentais para o projeto de arquitetura, porque permitem que alterações sejam localizadas. O padrão MVC, apresentado na Tabela 6.1, separa os elementos de um sistema, permitindo mudá-los de forma independente. Por exemplo, pode-se adicionar uma nova visão ou alterar uma exibição existente sem quaisquer alterações nos dados subjacentes do modelo. O padrão de arquitetura em camadas é outra maneira de conseguir a separação e independência. Esse padrão está na Tabela 6.2. Aqui, a funcionalidade do sistema é organizada em camadas separadas, e cada camada só depende dos recursos e serviços oferecidos pela camada imediatamente abaixo dela.

Figura 6.2 A organização do MVC

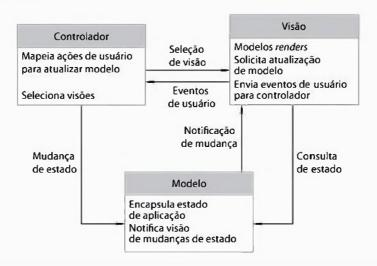
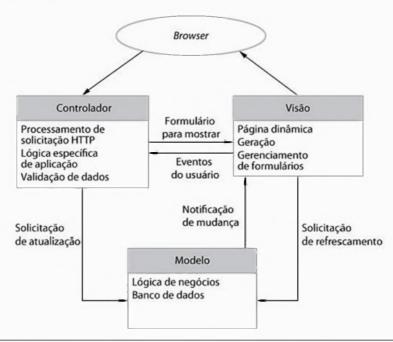



Figura 6.3 Arquitetura de aplicações Web usando o padrão MVC

Essa abordagem em camadas apoia o desenvolvimento incremental de sistemas. Quando uma camada é desenvolvida, alguns dos serviços prestados por ela podem ser disponibilizados para os usuários. A arquitetura também é mutável e portável. Enquanto sua interface for inalterada, uma camada pode ser substituída por outra equivalente. Além disso, quando a camada de interfaces muda ou tem novos recursos adicionados, apenas a camada adjacente é afetada. Como sistemas em camadas localizam dependências das máquinas em camadas internas, isso facilita o fornecimento de implementações de multiplataformas de um sistema de aplicação. Apenas as camadas internas dependentes da máquina precisam ser reimplementadas para levar em conta os recursos de um sistema operacional diferente ou banco de dados.

A Figura 6.4 é um exemplo de uma arquitetura em camadas, com quatro camadas. A camada mais baixa inclui software de apoio ao sistema — geralmente, apoio de banco de dados e de sistema operacional. A próxima camada é a camada de aplicação, que inclui os componentes relacionados com a funcionalidade da aplicação e os

Tabela 6.2 O padrão de arquitetura em camadas

Nome	Arquitetura em camadas
Descrição	Organiza o sistema em camadas com a funcionalidade relacionada associada a cada camada. Uma camada fornece serviços à camada acima dela; assim, os níveis mais baixos de camadas representam os principais serviços suscetíveis de serem usados em todo o sistema. Veja a Figura 6.4.
Exemplo	Um modelo em camadas de um sistema para compartilhar documentos com direitos autorais, em bibliotecas diferentes, como mostrado na Figura 6.5.
Quando é usado	É usado na construção de novos recursos em cima de sistemas existentes; quando o desenvolvimento está espalhado por várias equipes, com a responsabilidade de cada equipe em uma camada de funcionalidade; quando há um requisito de proteção multinível.
Vantagens	Desde que a interface seja mantida, permite a substituição de camadas inteiras. Recursos redundantes (por exemplo, autenticação) podem ser fornecidos em cada camada para aumentar a confiança do sistema.
Desvantagens	Na prática, costuma ser difícil proporcionar uma clara separação entre as camadas, e uma camada de alto nível pode ter de interagir diretamente com camadas de baixo nível, em vez de através da camada imediatamente abaixo dela. O desempenho pode ser um problema por causa dos múltiplos níveis de interpretação de uma solicitação de serviço, uma vez que são processados em cada camada.

Figura 6.4 Uma arquitetura genérica em camadas

Interface de usuário
Gerenciamento de interface de usuário Autenticação e autorização
Lógica de negócio principal/funcionalidade de aplicação Recursos de sistema
Apoio de sistema (SO, banco de dados etc.)

componentes utilitários que são usados por outros componentes da aplicação. A terceira camada está preocupada com o gerenciamento de interface de usuário e fornecimento de autenticação e autorização de usuário, com a camada superior fornecendo recursos de interface com o usuário. Certamente, o número de camadas é arbitrário. Qualquer camada na Figura 6.4 pode ser dividida em mais camadas.

A Figura 6.5 é um exemplo de como esse padrão de arquitetura em camadas pode ser aplicado a um sistema de biblioteca chamado LIBSYS, que permite controlar o acesso eletrônico de um grupo de bibliotecas universitárias aos materiais com direitos autorais. Esse sistema tem arquitetura de cinco camadas, na qual a camada inferior são os bancos de dados individuais de cada biblioteca.

Você pode ver um exemplo do padrão de arquitetura em camadas na Figura 6.12 (Seção 6.4). Ela mostra a organização do sistema de saúde mental (MHC-PMS) que discuti nos capítulos anteriores.

6.3.2 Arquitetura de repositório

A arquitetura em camadas e padrões MVC são exemplos de padrões nos quais a visão apresentada é a organização conceitual de um sistema. Meu próximo exemplo — o padrão Repositório (Tabela 6.3) — descreve como um conjunto de componentes que interagem podem compartilhar dados.

Figura 6.5 A arquitetura do sistema LIBSYS

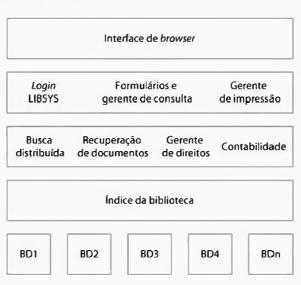



Tabela 6.3 O padrão repositório

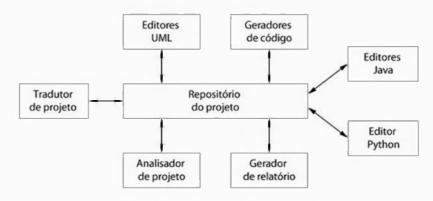
Nome	Repositório
Descrição	Todos os dados em um sistema são gerenciados em um repositório central, acessível a todos os componentes do sistema. Os componentes não interagem diretamente, apenas por meio do repositório.
Exemplo	A Figura 6.6 é um exemplo de um IDE em que os componentes usam um repositório de informações sobre projetos de sistema. Cada ferramenta de software gera informações que ficam disponíveis para uso por outras ferramentas.
Quando é usado	Você deve usar esse padrão quando tem um sistema no qual grandes volumes de informações são gerados e precisam ser armazenados por um longo tempo. Você também pode usá-lo em sistemas dirigidos a dados, nos quais a inclusão dos dados no repositório dispara uma ação ou ferramenta.
Vantagens	Os componentes podem ser independentes — eles não precisam saber da existência de outros componentes. As alterações feitas a um componente podem propagar-se para todos os outros. Todos os dados podem ser gerenciados de forma consistente (por exemplo, *backups* feitos ao mesmo tempo), pois tudo está em um só lugar.
Desvantagens	O repositório é um ponto único de falha, assim, problemas no repositório podem afetar todo o sistema. Pode haver ineficiências na organização de toda a comunicação através do repositório. Distribuir o repositório através de vários computadores pode ser difícil.

A maioria dos sistemas que usam grandes quantidades de dados é organizada em torno de um banco de dados ou repositório compartilhado. Esse modelo é, portanto, adequado para aplicações nas quais os dados são gerados por um componente e usados por outro. Exemplos desse tipo de sistema incluem sistemas de comando e controle, sistemas de informações gerenciais, sistemas de CAD e ambientes interativos de desenvolvimento de software.

A Figura 6.6 é uma ilustração de uma situação na qual um repositório pode ser usado. Esse diagrama mostra um IDE que inclui diversas ferramentas para apoiarem o desenvolvimento dirigido a modelos. O repositório, nesse caso, pode ser um ambiente controlado de versões (veja o Capítulo 25) que mantém o controle das alterações de software e permite a reversão para versões anteriores.

A organização de ferramentas em torno de um repositório constitui uma maneira eficiente de compartilhar grandes quantidades de dados. Não há necessidade de transmitir dados explicitamente de um componente para outro. No entanto, os componentes devem operar em torno de um modelo aprovado de repositório de dados. Inevitavelmente, esse é um compromisso entre as necessidades específicas de cada ferramenta, e pode ser difícil ou impossível integrar novos componentes se seus modelos de dados não se adequam ao esquema acordado. Na prática, pode ser difícil distribuir o repositório por meio de uma série de máquinas. Embora seja possível a distribuição de um repositório logicamente centralizado, pode haver problemas com a redundância e inconsistência de dados.

No exemplo mostrado na Figura 6.6, o repositório é passivo e o controle é de responsabilidade dos componentes que usam o repositório. Uma abordagem alternativa, derivada de sistemas de inteligência artificial, usa um mo-

Figura 6.6 Uma arquitetura de repositório para um IDE

delo 'quadro-negro' que aciona os componentes do modelo quando dados específicos se tornam disponíveis. Isso é apropriado quando a forma de repositório de dados não é tão bem estruturada. As decisões sobre qual ferramenta ativar só podem ser feitas quando os dados são analisados. Esse modelo foi introduzido por Nii (1986). Bosch (2000) incluiu uma boa discussão sobre como esse estilo se relaciona com atributos de qualidade do sistema.

6.3.3 Arquitetura cliente-servidor

O padrão repositório está preocupado com a estrutura estática de um sistema e não mostra sua organização de tempo de execução. Meu próximo exemplo ilustra uma organização muito usada para sistemas distribuídos, em tempo de execução. O padrão cliente-servidor é descrito na Tabela 6.4.

Um sistema que segue o padrão cliente-servidor é organizado como um conjunto de serviços e servidores associados e clientes que acessam e usam os serviços. Os principais componentes desse modelo são:

1. Um conjunto de servidores que oferecem serviços a outros componentes. Exemplos de servidores incluem: servidores de impressão que oferecem serviços de impressão; servidores de arquivos que oferecem serviços de gerenciamento de arquivos; e um servidor de compilação, que oferece serviços de compilação de linguagens de programação.
2. Um conjunto de clientes que podem chamar os serviços oferecidos pelos servidores. Em geral, haverá várias instâncias de um programa cliente executando simultaneamente em computadores diferentes.
3. Uma rede que permite aos clientes acessar esses serviços. A maioria dos sistemas cliente-servidor é implementada como sistemas distribuídos, conectados através de protocolos de Internet.

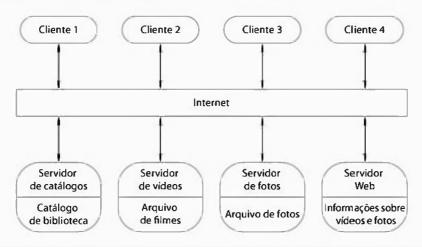
Arquiteturas cliente-servidor são normalmente consideradas arquiteturas de sistemas distribuídos, mas o modelo lógico de serviços independentes rodando em servidores separados pode ser implementado em um único computador. Novamente, um benefício importante é a separação e a independência. Serviços e servidores podem ser modificados sem afetar outras partes do sistema.

Os clientes podem ter de saber os nomes dos servidores disponíveis e os serviços que eles fornecem. No entanto, os servidores não precisam conhecer a identidade dos clientes ou quantos clientes estão acessando seus serviços. Os clientes acessam os serviços fornecidos por um servidor por meio de chamadas de procedimento remoto usando um protocolo de solicitação-resposta, tal como o protocolo HTTP usado na Internet. Essencialmente, um cliente faz uma solicitação a um servidor e espera até receber uma resposta.

A Figura 6.7 é um exemplo de um sistema baseado no modelo cliente-servidor. Esse é um sistema multiusuário baseado na Internet para fornecimento de uma biblioteca de filmes e fotos. Nesse sistema, vários servidores gerenciam e apresentam os diferentes tipos de mídia. Os quadros de vídeo precisam ser transmitidos de forma rápida e em sincronia, mas em resolução relativamente baixa. Eles podem ser comprimidos em um arquivo, de modo que o servidor de vídeo possa lidar com a compressão e descompressão de vídeo em diferentes formatos. As fotos, porém, devem permanecer em uma resolução alta; por isso, é conveniente mantê-las em um servidor separado.

Tabela 6.4 O padrão cliente-servidor

Nome	Cliente-servidor
Descrição	Em uma arquitetura cliente-servidor, a funcionalidade do sistema está organizada em serviços — cada serviço é prestado por um servidor. Os clientes são os usuários desses serviços e acessam os servidores para fazer uso deles.
Exemplo	A Figura 6.7 é um exemplo de uma biblioteca de filmes e vídeos/DVDs, organizados como um sistema cliente-servidor.
Quando é usado	É usado quando os dados em um banco de dados compartilhado precisam ser acessados a partir de uma série de locais. Como os servidores podem ser replicados, também pode ser usado quando a carga em um sistema é variável.
Vantagens	A principal vantagem desse modelo é que os servidores podem ser distribuídos através de uma rede. A funcionalidade geral (por exemplo, um serviço de impressão) pode estar disponível para todos os clientes e não precisa ser implementada por todos os serviços.
Desvantagens	Cada serviço é um ponto único de falha suscetível a ataques de negação de serviço ou de falha do servidor. O desempenho, bem como o sistema, pode ser imprevisível, pois depende da rede. Pode haver problemas de gerenciamento se os servidores forem propriedade de diferentes organizações.

Figura 6.7 Uma arquitetura cliente-servidor para uma biblioteca de filmes

O catálogo deve ser capaz de lidar com uma variedade de consultas e fornecer links para o sistema de informação da Internet que incluam dados sobre os filmes e videoclipes, além de um sistema de comércio eletrônico que apoie a venda das fotografias, filmes e videoclipes. O programa do cliente é, simplesmente, uma interface de usuário integrada, construída usando-se um *browser* da Internet para acesso a esses serviços.

A principal vantagem do modelo cliente-servidor é que se trata de uma arquitetura distribuída. O uso efetivo pode ser feito por sistemas em rede com muitos processadores distribuídos. É fácil adicionar um novo servidor e integrá-lo com o resto do sistema ou atualizar os servidores de forma transparente, sem afetar outras partes do sistema. No Capítulo 18, discuto as arquiteturas distribuídas, incluindo arquiteturas cliente-servidor e arquiteturas de objetos distribuídos.

6.3.4 Arquitetura de duto e filtro

Meu último exemplo de um padrão de arquitetura é o padrão duto e filtro. Esse é um modelo de organização em tempo de execução de um sistema no qual as transformações funcionais processam suas entradas e produzem saídas. Os dados fluem de um para o outro e transformam-se enquanto se movem através da sequência. Cada etapa do processamento é implementada como uma transformação. Os dados de entrada fluem por meio dessas transformações até serem convertidos em saídas. As transformações podem executar sequencialmente ou em paralelo. Os dados podem ser processados por cada item de transformação ou em um único lote.

Tabela 6.5 O padrão duto e filtro

Nome	Duto e filtro
Descrição	O processamento dos dados em um sistema está organizado de modo que cada componente de processamento (filtro) seja discreto e realize um tipo de transformação de dados. Os dados fluem (como em um duto) de um componente para outro para processamento.
Exemplo	A Figura 6.8 é um exemplo de um sistema de duto e filtro usado para o processamento das faturas.
Quando é usado	Comumente, é usado em aplicações de processamento de dados (tanto as baseadas em lotes como as baseadas em transações) em que as entradas são processadas em etapas separadas para gerarem saídas relacionadas.
Vantagens	O reúso da transformação é de fácil compreensão e suporte. Estilo de *workflow* corresponde à estrutura de muitos processos de negócios. Evolução por adição de transformações é simples. Pode ser implementado tanto como um sistema sequencial quanto concorrente.
Desvantagens	O formato para transferência de dados tem de ser acordado entre as transformações de comunicação. Cada transformação deve analisar suas entradas e gerar as saídas para um formato acordado. Isso aumenta o *overhead* do sistema e pode significar a impossibilidade de reúso de transformações funcionais que usam estruturas incompatíveis de dados.

O nome 'duto e filtro' vem do sistema original Unix, em que foi possível vincular os processos usando 'dutos'. Estes passam um fluxo de texto de um processo para outro. Os sistemas que obedecem a esse modelo podem ser implementados por meio da combinação de comandos Unix, usando dutos e recursos de controle do *shell* do Unix. O termo 'filtro' é usado porque uma transformação 'filtra' os dados que ele pode processar a partir de seu fluxo de dados de entrada (veja a Tabela 6.5).

Variantes desse padrão têm sido usadas desde que os computadores começaram a ser usados para o processamento automático de dados. Quando as transformações são sequenciais com dados transformados em lotes, esse modelo de arquitetura de duto e filtro torna-se um modelo sequencial, uma arquitetura comum para sistemas de processamento de dados (por exemplo, um sistema de faturamento). A arquitetura de um sistema embutido pode também ser organizada como um duto de processo, com cada processo em execução concorrente. No Capítulo 20, eu discuto o uso desse padrão em sistemas embutidos.

Um exemplo desse tipo de arquitetura de sistema, usado em um aplicativo de processamento em lote, é mostrado na Figura 6.8. Uma organização emitiu faturas para os clientes. Uma vez por semana, os pagamentos realizados são reconciliados com as faturas. Para cada pedido pago, um recibo é emitido. Para as faturas que não foram pagas dentro do prazo de pagamento permitido, é emitido um lembrete.

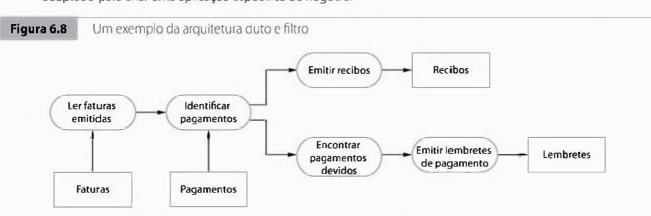
É difícil escrever os sistemas interativos usando o modelo de duto e filtro, por causa da necessidade de processamento dos fluxos de dados. Embora entradas e saídas de texto simples possam ser modeladas dessa forma, interfaces gráficas de usuário têm formatos E/S mais complexos e uma estratégia de controle baseada em eventos como cliques do mouse ou seleções de menus. É difícil traduzir isso de forma compatível com o modelo de duto.

6.4 Arquiteturas de aplicações

Sistemas de aplicação são destinados a atender uma necessidade organizacional ou de negócio. Todas as empresas têm muito em comum — elas precisam contratar pessoas, emitir faturas, manter a contabilidade, e assim por diante. As empresas que operam no mesmo setor usam aplicações específicas comuns. Portanto, assim como funções de negócios em geral, todas as empresas de telefonia precisam de sistemas para conectar chamadas, gerenciar sua rede, emitir faturas para os clientes etc. Assim, os sistemas de aplicação usados por essas empresas também têm muito em comum.

Essas semelhanças levaram ao desenvolvimento de arquiteturas de software que descrevem a estrutura e organização dos diferentes tipos de sistemas de software. Arquiteturas de aplicação encapsulam as principais características de uma classe de sistemas. Por exemplo, em sistemas de tempo real, pode haver modelos genéricos de arquitetura de diferentes tipos de sistemas, como sistemas de coleta de dados ou sistemas de monitoração. Embora instâncias desses sistemas difiram em detalhes, a estrutura comum de arquitetura pode ser reusada no desenvolvimento de novos sistemas do mesmo tipo.

A arquitetura de aplicação pode ser reimplementada no desenvolvimento de novos sistemas, mas, para sistemas de muitas empresas, o reúso de aplicações é possível sem a reimplementação. Vemos isso no crescimento da Enterprise Resource Planning (ERP) de empresas como SAP e Oracle, e pacotes de software verticais (COTS) para aplicações especializadas em diferentes áreas de negócio. Nesses sistemas, um sistema genérico é configurado e adaptado para criar uma aplicação específica de negócio.

Figura 6.8 Um exemplo da arquitetura duto e filtro

Por exemplo, um sistema de gerenciamento da cadeia de suprimentos pode ser adaptado para diferentes tipos de fornecedores, produtos e disposições contratuais.

Como um projeto de software, você pode usar modelos de arquiteturas de aplicações de inúmeras maneiras:

1. *Como ponto de partida para o processo de projeto de arquitetura.* Se você não estiver familiarizado com o tipo de aplicação que está desenvolvendo, pode basear seu projeto inicial em uma arquitetura genérica de aplicação. Naturalmente, ela precisará ser especializada para o sistema específico a ser desenvolvido, mas é um bom ponto de partida para o projeto.
2. *Como um checklist de projeto.* Se você desenvolveu um projeto de arquitetura para um sistema aplicativo, é possível compará-lo com a arquitetura de aplicação genérica. Você pode verificar se seu projeto é compatível com a arquitetura genérica.
3. *Como forma de organizar o trabalho da equipe de desenvolvimento.* As arquiteturas de aplicação identificam características estruturais estáveis das arquiteturas do sistema, e, em muitos casos, é possível desenvolvê-las em paralelo. Você pode distribuir o trabalho aos membros da equipe para implementação dos diferentes componentes dentro da arquitetura.
4. *Como uma forma de avaliar os componentes para reúso.* Se você tem componentes que possam ser reusados, você pode compará-los com as estruturas genéricas para ver se existem componentes comparáveis na arquitetura da aplicação.
5. *Como um vocabulário para falar sobre os tipos de aplicações.* Se você está discutindo uma aplicação específica ou tentando comparar as aplicações do mesmo tipo, então pode usar os conceitos identificados na arquitetura genérica para falar sobre as aplicações.

Existem muitos tipos de sistema de aplicação e, em alguns casos, eles podem parecer muito diferentes uns dos outros. No entanto, muitas dessas aplicações superficialmente diferentes têm, na verdade, muito em comum, e, portanto, podem ser representadas por uma arquitetura abstrata única de aplicação. Ilustro isso aqui, descrevendo as seguintes arquiteturas de dois tipos de aplicação:

1. *Aplicações de processamento de transações.* São aplicações centradas em banco de dados que processam os pedidos do usuário para obterem informações e atualizarem as informações em um banco de dados. Consistem no tipo mais comum de sistemas interativos de negócios. Elas são organizadas de tal forma que as ações do usuário não podem interferir umas com as outras e a integridade do banco de dados é mantida. Essa classe de sistema inclui sistemas bancários interativos, sistemas de comércio eletrônico, sistemas de informação e sistemas de reservas.
2. *Sistemas de processamento de linguagens.* São sistemas nos quais as intenções do usuário são expressas em uma linguagem formal (como Java). O sistema de processamento de linguagem processa essa linguagem em um formato interno e interpreta essa representação interna. Os mais conhecidos sistemas de processamento de linguagens são os compiladores, que traduzem programas em linguagem de alto nível em código de máquina. No entanto, os sistemas de processamento de linguagens também são usados para interpretar linguagens de comando para bancos de dados e sistemas de informação, bem como linguagens de marcação como XML (HAROLD e MEANS, 2002; HUNTER et al., 2007).

Escolhi esses tipos particulares de sistema, pois um grande número de sistemas de negócios baseados em Internet são sistemas de processamento de transações, e todo o desenvolvimento de software baseia-se em sistemas de processamento de linguagens.

6.4.1 Sistemas de processamento de transações

Sistemas de processamento de transações (TP, do inglês *transaction processing*) são projetados para processar os pedidos de informação do usuário de um banco de dados ou pedidos para atualizar um banco de dados (LEWIS et al., 2003). Tecnicamente, uma transação é uma sequência de operações tratadas como uma única unidade (uma unidade atômica). Todas as operações em uma transação devem ser concluídas antes da mudança de banco de dados se tornar permanente. Isso garante que a falha de operações dentro da transação não gere inconsistências no banco de dados.

Do ponto de vista do usuário, uma transação é uma sequência de ações coerentes que satisfazem uma meta, como 'encontrar os tempos de voo de Londres para Paris'. Se a transação de usuário não necessita de alteração no banco de dados, então pode não ser necessário empacotar essa transação como uma transação técnica de banco de dados.

Um exemplo de uma transação é um pedido de um cliente para retirar dinheiro de uma conta bancária usando um caixa eletrônico (ATM — *Automated Teller Machine*). Trata-se de obter detalhes da conta do cliente, verificando o saldo e modificando-o com a retirada de uma quantia, e enviar comandos ao ATM para a entrega do dinheiro. Até que todas essas etapas sejam concluídas, a transação é incompleta e o banco de dados de contas do cliente não é alterado.

Sistemas de processamento de transações são normalmente interativos, em que os usuários fazem solicitações assíncronas por serviço. A Figura 6.9 ilustra a estrutura conceitual da arquitetura de aplicações TP. Primeiro, um usuário faz uma solicitação ao sistema por meio de um componente de E/S de processamento. O pedido é processado por alguma lógica específica de aplicação. Uma transação é criada e passada para um gerenciador de transações, que geralmente é embutido no sistema de gerenciamento de banco de dados. Depois que o gerenciador de transações garantiu que a transação seja devidamente completada, ele sinaliza para a aplicação que o processamento terminou.

Os sistemas de processamento de transações podem ser organizados com uma arquitetura de 'duto e filtro' com componentes do sistema responsável pelas entradas, processamento e saídas. Por exemplo, considere um sistema bancário que permite aos clientes consultar suas contas e sacar dinheiro em caixas eletrônicos. O sistema é composto de dois componentes de software que se colaboram, o software de caixa eletrônico e o de processamento de conta no servidor do banco de dados. Os componentes de entrada e saída são implementados como software no caixa eletrônico, e o componente de processamento é parte do servidor do banco de dados. A Figura 6.10 mostra a arquitetura desse sistema, ilustrando as funções dos compontentes de entrada, processo e saída.

6.4.2 Os sistemas de informação

Todos os sistemas que envolvem a interação com bancos de dados compartilhados podem ser considerados sistemas de informação baseados em transações. Um sistema de informação permite acesso controlado a uma grande base de informações, como um catálogo de biblioteca, um horário de voo ou os registros de pacientes em um hospital. Cada vez mais, sistemas de informação são sistemas baseados na Internet, acessados por meio de um *browser*.

A Figura 6.11 é um modelo geral de um sistema de informação. O sistema é modelado usando-se uma abordagem em camadas (discutida na Seção 6.3), na qual a camada superior apoia a interface do usuário, e a inferior é o banco de dados do sistema. A camada de comunicação de usuário lida com todas as entradas e saídas da interface de usuário, e a camada de recuperação de informação inclui uma lógica específica de aplicação para acessar e atualizar o banco de dados. Como veremos mais tarde, as camadas desse modelo podem mapear diretamente os servidores em um sistema baseado na Internet.

Como exemplo de uma instanciação do modelo em camadas, a Figura 6.12 mostra a arquitetura do MHC-PMS. Lembre-se de que esse sistema mantém e gerencia dados de pacientes que estão consultando médicos especialistas sobre problemas de saúde mental. Acrescentei detalhes em cada camada do modelo, identificando os componentes que oferecem apoio a comunicações de usuário e recuperação e acesso de informação:

1. A camada superior é responsável pela implementação da interface de usuário. Nesse caso, a interface foi implementada usando-se um *browser*.
2. A segunda camada fornece a funcionalidade de interface de usuário que é entregue por meio do *browser*. Ela inclui componentes que permitem aos usuários fazerem o *login* no sistema e componentes de verificação que asseguram que as operações usadas pelos usuários sejam permitidas para seu papel. Essa camada inclui formulários e componentes de gerenciamento de menu que apresentam informações aos usuários, além dos componentes de validação de dados que verificam a consistência das informações.

Figura 6.9 A estrutura de aplicações de processamento de transações

Figura 6.10 A arquitetura de software de um sistema de ATM

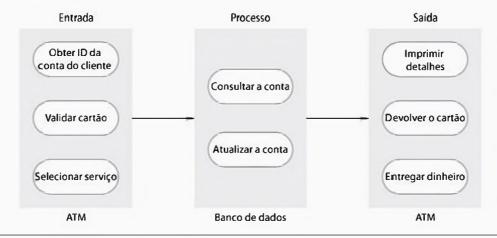



Figura 6.11 Arquitetura de sistema de informação em camadas

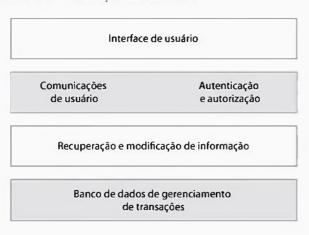
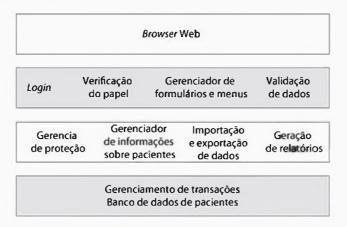



Figura 6.12 A arquitetura do MHC-PMS

3. A terceira camada define a funcionalidade do sistema e fornece componentes que implementam a proteção do sistema, criação e atualização de informações de pacientes, importação e exportação dos dados de pacientes a partir de outros bancos de dados, além dos geradores de relatório que criam relatórios de gerenciamento.

4. Finalmente, a camada mais baixa, construída usando um sistema de gerenciamento de banco de dados comercial, fornece gerenciamento de transações e repósitório persistente de dados.

Normalmente, sistemas de informação e de gerenciamento de recursos são baseados na Web, nos quais as interfaces de usuário são implementadas usando-se um *browser*. Por exemplo, sistemas de comércio eletrônico são sistemas de gerenciamento de recursos baseados na Internet que aceitam pedidos eletrônicos de bens ou serviços e, em seguida, providenciam a entrega desses bens ou serviços ao cliente. Em um sistema de comércio eletrônico, a camada específica de aplicação inclui funcionalidade adicional que apoia um 'carrinho de compras', no qual os usuários podem colocar uma série de itens em transações separadas, e em seguida, pagar por todos juntos em uma única transação.

Nesses sistemas, a organização dos servidores geralmente reflete o modelo genérico de quatro camadas apresentado na Figura 6.11. Esses sistemas são frequentemente implementados como arquiteturas/cliente/servidor multicamadas, conforme discutido no Capítulo 18:

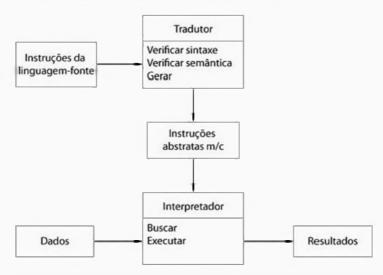
1. O servidor Web é responsável por todas as comunicações de usuário, com a interface de usuário implementada usando um *browser*;
2. O servidor de aplicações é responsável por implementar a lógica específica de aplicação, bem como o repositório de informações e pedidos de recuperação;
3. O servidor do banco de dados move as informações para e a partir do banco de dados, e lida com o gerenciamento de transações.

Usar vários servidores permite que se tenha alto desempenho e faz com que seja possível lidar com centenas de transações por minuto. À medida que a demanda aumenta, os servidores podem ser adicionados em cada nível para lidar com o processamento extra envolvido.

6.4.3 Sistemas de processamento de linguagens

Os sistemas de processamento de linguagens traduzem uma linguagem natural ou artificial em outra representação dessa linguagem. Para linguagens de programação, também podem executar o código resultante. Em engenharia de software, compiladores traduzem uma linguagem artificial de programação em código de máquina. Outros sistemas de processamento de linguagens podem traduzir uma descrição XML de dados em comandos, para consultar um banco de dados ou uma representação XML alternativa. Sistemas de processamento da linguagem natural podem traduzir uma linguagem natural para outra, por exemplo, o francês para o norueguês.

Na Figura 6.13, é ilustrada uma possível arquitetura para um sistema de processamento de linguagem para uma linguagem de programação. As instruções da linguagem-fonte definem o programa a ser executado, e um tradutor converte as instruções para uma máquina abstrata. Essas instruções são, então, interpretadas por outro

Figura 6.13 A arquitetura de um sistema de processamento de linguagem

componente, que busca as instruções para a execução e as executa usando (se necessário) os dados do ambiente. A saída do processo é o resultado da interpretação das instruções sobre os dados de entrada.

Claro que, para muitos compiladores, o interpretador é uma unidade de hardware que processa as instruções de máquina, e a máquina abstrata é um processador real. No entanto, para linguagens tipadas dinamicamente, como Python, o interpretador pode ser um componente de software.

Os compiladores de linguagem de programação que são parte de um ambiente de programação mais geral têm uma arquitetura genérica (Figura 6.14) que inclui os seguintes componentes:

1. Um analisador léxico, que toma os *tokens* da linguagem de entrada e os converte em um formato interno.
2. Uma tabela de símbolos, que contém informações sobre os nomes das entidades (variáveis, nomes de classes, nomes de objetos etc.) usadas no texto que está sendo traduzido.
3. Um analisador sintático, que verifica a sintaxe da linguagem a ser traduzida. Ele usa uma gramática definida da linguagem e constrói uma árvore de sintaxe.
4. Uma árvore de sintaxe, que é uma estrutura interna que representa o programa a ser compilado.
5. Um analisador semântico, que usa informações da árvore de sintaxe e a tabela de símbolos para verificar a correção semântica do texto na linguagem de entrada.
6. Um gerador de código que 'anda' na árvore de sintaxe e gera códigos de máquina abstrata.

Também podem ser incluídos outros componentes que analisam e transformam a árvore de sintaxe para melhorar a eficiência e eliminar a redundância do código de máquina gerado. Em outros tipos de sistema de processamento de linguagem, como um tradutor de linguagem natural, haverá componentes adicionais, como um dicionário, e o código gerado é a entrada de texto traduzido para outra linguagem.

Existem padrões de arquitetura alternativos que podem ser usados em um sistema de processamento da linguagem (GARLAN e SHAW, 1993). Os compiladores podem ser implementados usando um composto de um repositório e um modelo de duto e filtro. Em uma arquitetura de compilador, a tabela de símbolos é um repositório de dados compartilhados. As fases da análise léxica, sintática e semântica estão organizadas em sequência, como mostra a Figura 6.14, e comunicam-se por meio da tabela de símbolos compartilhados.

Esse modelo de duto e filtro de compilação de linguagem é eficaz em ambientes em lotes nos quais os programas são compilados e executados sem interação do usuário — por exemplo, na tradução de um documento XML para outro. É menos efetivo quando um compilador é integrado com outras ferramentas de processamento de linguagem, como um sistema de edição estruturado, um depurador interativo ou um *prettyprinter* (formatador de impressão) de programa. Nessa situação, as mudanças de um componente precisam refletir imediatamente em outros componentes. Portanto, é melhor organizar o sistema em torno de um repositório, como se vê na Figura 6.15.

Essa figura ilustra como um sistema de processamento de linguagem pode ser parte de um conjunto integrado de ferramentas de apoio à programação. Nesse exemplo, a tabela de símbolos e a árvore de sintaxe funcionam como um repositório central de informações. Ferramentas ou fragmentos de ferramentas comunicam-se por meio dele. Outras informações que às vezes são embutidas em ferramentas, como a definição da gramática e a definição do formato de saída para o programa, foram retiradas das ferramentas e colocadas no repositório. Portanto, um editor dirigido à sintaxe pode verificar se a sintaxe de um programa está correta, uma vez que está sendo digitado e um *prettyprinter* pode criar listagens de programa em um formato que seja fácil de ler.

Figura 6.14 Arquitetura do compilador em duto e filtro

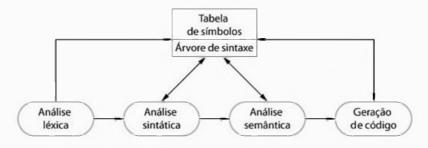
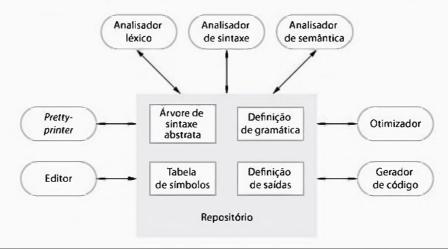



Figura 6.15 Uma arquitetura de repositório para um sistema de processamento de linguagem

PONTOS IMPORTANTES

- Uma arquitetura de software é uma descrição de como um sistema de software é organizado. As propriedades de um sistema, como desempenho, proteção e disponibilidade, são influenciadas pela arquitetura adotada.
- As decisões de projeto de arquitetura incluem decisões sobre o tipo de aplicação, a distribuição do sistema, os estilos da arquitetura a serem utilizados e as formas como a arquitetura deve ser documentada e avaliada.
- As arquiteturas podem ser documentadas a partir de diferentes perspectivas ou visões. As possíveis visões incluem uma visão conceitual, uma visão lógica, uma visão de processo, uma visão de desenvolvimento e uma visão física.
- Os padrões da arquitetura são um meio de reusar o conhecimento sobre as arquiteturas genéricas de sistemas. Eles descrevem a arquitetura, explicam quando elas podem ser usadas e discutem suas vantagens e desvantagens.
- Entre os padrões de arquitetura comumente usados estão: Modelo-Visão-Controlador, Arquitetura em camadas, Repositório, Cliente-servidor e Duto e filtro.
- Modelos genéricos de arquiteturas de sistemas de aplicação ajudam-nos a compreender o funcionamento das aplicações, comparar aplicações do mesmo tipo, validar projetos de sistemas de aplicação e avaliar os componentes para reúso em larga escala.
- Sistemas de processamento de transações são sistemas interativos que permitem que as informações em um banco de dados sejam acessadas remotamente e modificadas por vários usuários. Os sistemas de informação e de gerenciamento de recursos são exemplos de sistemas de processamento de transações.
- Sistemas de processamento de linguagens são usados para traduzir textos de uma linguaguem para outra e para realizar as instruções especificadas em uma linguagem de entrada. Eles incluem um tradutor e uma máquina abstrata que executa a linguagem gerada.

LEITURA COMPLEMENTAR

Software Architecture: Perspectives on an Emerging Discipline. Esse foi o primeiro livro sobre arquitetura de software, apresenta uma boa discussão sobre diferentes estilos de arquitetura. (SHAW, M.; GARLAN, D. *Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall*, 1996.)

Software Architecture in Practice, 2nd ed. Essa é uma discussão prática sobre arquiteturas de software que não exagera nos benefícios do projeto de arquitetura. O livro fornece uma lógica clara de negócio, explicando por que as arquiteturas são importantes. (BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 2. ed. Addison-Wesley, 2003.)

The Golden Age of Software Architecture. Esse trabalho avalia o desenvolvimento da arquitetura de software desde seu início, na década de 1980, até seu uso atual. Contém pouco conteúdo técnico, mas apresenta um interessante panorama histórico. (SHAW, M.; CLEMENTS, P. The Golden Age of Software Architecture. *IEEE Software*, v. 21, n. 2, mar.-abr. 2006.) Disponível em: <http://dx.doi.org/10.1109/MS.2006.58>.

Handbook of Software Architecture. Esse é um trabalho em andamento realizado por Grady Booch, um dos primeiros evangelizadores da arquitetura de software. Ele vem documentando as arquiteturas de vários sistemas de software para que você possa ver a realidade em vez de abstrações acadêmicas. Disponível em: <http://www.handbookofsoftwarearchitecture.com/>, e destina-se a aparecer como um livro.

EXERCÍCIOS

6.1 Ao descrever um sistema, explique por que você pode precisar projetar sua arquitetura antes de a especificação de requisitos estar completa.

6.2 Você foi convidado para preparar e entregar uma apresentação para um gerente não técnico, a fim de justificar a contratação de um arquiteto de sistemas para um novo projeto. Escreva uma lista de aspectos que definam os pontos principais de sua apresentação. Naturalmente, você deve explicar o que se entende por arquitetura de sistema.

6.3 Explique por que podem surgir conflitos ao se projetar uma arquitetura na qual tanto os requisitos quanto a disponibilidade e a proteção sejam os mais importantes requisitos não funcionais.

6.4 Desenhe diagramas mostrando uma visão conceitual e uma visão de processo das arquiteturas dos sistemas a seguir:

- Um sistema automatizado de emissão de bilhetes, usado por passageiros em uma estação ferroviária.
- Um sistema de videoconferência controlado por computador, que permite que áudio, vídeo e dados de computador sejam visíveis para vários participantes ao mesmo tempo.
- Um robô limpador de chão que se destina a limpar os espaços relativamente livres, como corredores. O robô deve ser capaz de perceber paredes e outras obstruções.

6.5 Explique por que, ao projetar a arquitetura de um sistema de grande porte, você normalmente usa vários padrões de arquitetura. Além das informações sobre os padrões discutidos neste capítulo, quais informações adicionais podem ser úteis no projeto de sistemas de grande porte?

6.6 Sugira uma arquitetura para um sistema (tal qual o iTunes) que sirva para vender e distribuir música na Internet. Que padrões são a base para essa arquitetura?

6.7 Explique como você usaria o modelo de referência de ambientes CASE (disponíveis no site do livro na Internet) para comparar as IDEs oferecidas por diferentes fornecedores de uma linguagem de programação como a Java.

6.8 Usando o modelo genérico de um sistema de processamento de linguagem aqui apresentado, projete a arquitetura de um sistema que aceita comandos de linguagem natural e traduz esses comandos em consultas de bancos de dados em uma linguagem como SQL.

6.9 Usando o modelo básico de um sistema de informação conforme apresentado na Figura 6.11, sugira os componentes que podem fazer parte de um sistema de informação que permite aos usuários visualizarem informações sobre os voos que chegam e partem de um aeroporto particular.

6.10 Deveria haver uma profissão específica, de 'arquiteto de software', cujo papel seria trabalhar com o cliente, de forma independente, para projetar a arquitetura do sistema de software? Uma empresa de software independente implementaria o sistema. Quais poderiam ser as dificuldades de se estabelecer essa profissão?

REFERÊNCIAS

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 2. ed. Boston: Addison-Wesley, 2003.

BERCZUK, S. P.; APPLETON, B. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston: Addison-Wesley, 2002.

BOOCH, G. *Handbook of software architecture*. Publicação Web. Disponível em: <http://www.handbookofsoftwarearchitecture.com>. 2009.

BOSCH, J. *Design and Use of Software Architectures*. Harlow, Reino Unido: Addison-Wesley, 2000.

BUSCHMANN, F.; HENNEY, K.; SCHMIDT, D. C. Pattern-oriented Software Architecture. v. 4. *A Pattern Language for Distributed Computing*. Nova York: John Wiley & Sons, 2007a.

______. Pattern-oriented Software Architecture. v. 5. *On Patterns and Pattern Languages*. Nova York: John Wiley & Sons, 2007b.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P. Pattern-oriented Software Architecture. v. 1. *A System of Patterns*. Nova York: John Wiley & Sons, 1996.

CLEMENTS, P.; BACHMANN, F.; BASS, L.; GARLAN, D.; IVERS, J.; LITTLE, R. et al. *Documenting Software Architectures: Views and Beyond*. Boston: Addison-Wesley, 2002.

COPLIEN, J. H.; HARRISON, N. B. *Organizational Patterns of Agile Software Development*. Englewood Cliffs, NJ: Prentice Hall, 2004.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.

GARLAN, D.; SHAW, M. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, v. 1, p. 1-39, 1993.

HAROLD, E. R.; MEANS, W. S. *XML in a Nutshell*. Sebastopol. Califórnia: O'Reilly, 2002.

HOFMEISTER, C.; NORD, R.; Soni, D. *Applied Software Architecture*. Boston: Addison- Wesley, 2000.

HUNTER, D.; RAFTER, J.; FAWCETT, J.; VAN DER VLIST, E. *Beginning XML*. 4. ed. Indianapolis, Ind.: Wrox Press, 2007.

KIRCHER, M.; JAIN, P. Pattern-Oriented Software Architecture. v. 3. *Patterns for Resource Management*. Nova York: John Wiley & Sons, 2004.

KRUCHTEN, P. The 4 + 1 view model of software architecture. *IEEE Software*, v. 12, n. 6, p. 42-50, 1995.

LANGE, C. F. J.; CHAUDRON, M. R. V.; MUSKENS, J. UML software description and architecture description. *IEEE Software*, v. 23, n. 2, p. 40-46, 2006.

LEWIS, P. M.; BERNSTEIN, A. J.; KIFER, M. *Databases and Transaction Processing: An Application-oriented Approach*. Boston: Addison-Wesley, 2003.

MARTIN, D.; SOMMERVILLE, I. Patterns of interaction: Linking ethnomethodology and design. *ACM Trans. on Computer-Human Interaction*, v. 11, n. 1, p. 59-89, 2004.

NII, H. P. Blackboard systems, parts 1 and 2. *AI Magazine*, v. 7, n. 3, 4, p. 38-53, 62-69, 1986.

SCHMIDT, D.; STAL, M.; ROHNERT, H.; BUSCHMANN, F. Pattern-Oriented Software Architecture. v. 2. *Patterns for Concurrent and Networked Objects*. Nova York: John Wiley & Sons, 2000.

SHAW, M.; GARLAN, D. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall, 1996.

USABILITY GROUP. *Usability patterns*. Publicação Web. Disponível em: <http://www.it.bton.ac.uk/cil/usability/patterns>. 1998.

CAPÍTULO

1 2 3 4 5 6 **7** 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Projeto e implementação

Objetivos

Os objetivos deste capítulo são apresentar o projeto de software orientado a objetos usando a UML e destacar os interesses importantes de implementação. Com a leitura deste capítulo, você:

- compreenderá as atividades mais importantes em um processo geral de projeto orientado a objetos;
- compreenderá alguns dos diferentes modelos que podem ser usados para documentar um projeto orientado a objetos;
- conhecerá a ideia de padrões de projeto e como eles são uma forma de reusar conhecimento e experiência de projeto;
- terá sido apresentado a questões fundamentais que precisam ser consideradas na implementação do software, incluindo o reúso de software e desenvolvimento *open source*.

Conteúdo

O projeto e implementação de software é um estágio do processo no qual um sistema de software executável é desenvolvido. Para alguns sistemas simples, o projeto e implementação de software é a engenharia de software, e todas as outras atividades são intercaladas com esse processo. No entanto, para grandes sistemas, o projeto e implementação de software é apenas parte de um conjunto de processos (engenharia de requisitos, verificação e validação etc.) envolvidos na engenharia de software.

As atividades de projeto e implementação de software são invariavelmente intercaladas. O projeto de software é uma atividade criativa em que você identifica os componentes de software e seus relacionamentos com base nos requisitos do cliente. A implementação é o processo de concretização do projeto como um programa. Às vezes, existe um estágio de projeto separado e esse projeto é modelado e documentado. Em outras ocasiões, um projeto está 'na cabeça' do programador ou esboçado em um quadro ou em papel. Um projeto trata de como resolver um problema, por isso, sempre existe um processo de projeto. No entanto, nem sempre é necessário ou apropriado descrever o projeto em detalhes usando a UML ou outra linguagem de descrição.

O projeto e a implementação estão intimamente ligados e, ao elaborar um projeto, você deve levar em consideração os problemas de implementação. Por exemplo, usar a UML para documentar um projeto pode ser a coisa certa a fazer se você estiver programando em uma linguagem orientada a objetos como Java ou C#. Mas é menos útil, penso eu, se você estiver desenvolvendo em uma linguagem com tipagem dinâmica como Python, e não faz sentido se você estiver imple-

mentando seu sistema, configurando um pacote de prateleira. Como discutido no Capítulo 3, os métodos ágeis normalmente funcionam a partir de esboços informais do projeto e deixam muitas decisões para os programadores.

Uma das decisões de implementação mais importantes que precisa ser tomada em um estágio inicial de um projeto de software é se você deve ou não comprar ou construir o software de aplicação. Atualmente, é possível comprar sistemas de prateleira (COTS, do inglês *commercial off-the-shelf*) em uma ampla variedade de domínios. Os sistemas podem ser adaptados e ajustados aos requisitos dos usuários. Por exemplo, se você quiser implementar um sistema de prontuário médico, você pode comprar um pacote que já seja usado em hospitais. Essa abordagem pode ser mais barata e rápida do que desenvolver um sistema em uma linguagem de programação convencional.

Quando você desenvolve uma aplicação dessa forma, o processo de projeto preocupa-se em 'como' usar os recursos de configuração desse sistema para cumprir os requisitos do sistema. Usualmente, você não desenvolve modelos de projeto do sistema, como modelos de objetos do sistema e suas interações. No Capítulo 16 discuto essa abordagem para o desenvolvimento baseado em COTS.

Presumo que a maioria dos leitores deste livro já tem experiência em projeto e implementação de programas. Isso é algo que você adquire quando aprende a programar e dominar os elementos de uma linguagem de programação como Java ou Python. Você provavelmente já aprendeu as boas práticas da programação nas linguagens de programação que estudou, bem como a forma de depurar os programas que desenvolveu. Portanto, não tratarei dos tópicos de programação aqui. Em vez disso, este capítulo tem dois objetivos:

1. Mostrar como a modelagem de sistema e o projeto de arquitetura (tratados nos capítulos 5 e 6) são colocados em prática no desenvolvimento de um projeto de software orientado a objetos.
2. Apresentar questões importantes de implementação que geralmente não são discutidas em livros de programação. Estas incluem o reúso de software, o gerenciamento de configuração e o desenvolvimento *open source*.

Como existe um grande número de plataformas de desenvolvimento diferentes, este capítulo não é voltado para uma linguagem de programação ou tecnologia de implementação específica. Portanto, apresento todos os exemplos usando a UML em vez de uma linguagem de programação como Java ou Python.

7.1 Projeto orientado a objetos com UML

Um sistema orientado a objetos é composto de objetos interativos que mantêm seu próprio estado local e oferecem operações nesse estado. A representação do estado é privada e não pode ser acessada diretamente, de fora do objeto. Processos de projeto orientado a objetos envolvem projetar as classes de objetos e os relacionamentos entre essas classes. Essas classes definem os objetos no sistema e suas interações. Quando o projeto é concebido como um programa em execução, os objetos são criados dinamicamente a partir dessas definições de classe.

Sistemas orientados a objetos são mais fáceis de mudar do que os sistemas desenvolvidos com abordagens funcionais. Os objetos incluem os dados e as operações para manipulá-los. Portanto, eles podem ser entendidos e modificados como entidades autônomas. Alterar a implementação de um objeto ou adicionar serviços não deve afetar outros objetos do sistema. Como os objetos são associados com coisas, muitas vezes existe um mapeamento claro entre entidades do mundo real (como componentes de hardware) e seus objetos de controle no sistema, o que melhora a inteligibilidade e, portanto, a manuteniblidade do projeto.

Para desenvolver um projeto de um sistema desde o conceito até o projeto detalhado orientado a objeto, existem várias atitudes que você precisa tomar:

1. Compreender e definir o contexto e as interações externas com o sistema.
2. Projetar a arquitetura do sistema.
3. Identificar os principais objetos do sistema.
4. Desenvolver modelos de projeto.
5. Especificar interfaces.

Como todas as atividades criativas, o projeto não é um processo sequencial claro. Você desenvolve um projeto tendo ideias, propondo soluções e refinando essas soluções assim que as informações ficam disponíveis. Quando os problemas surgem, inevitavelmente você tem de voltar atrás e tentar novamente. Às vezes, você explora as opções detalhadamente para ver se elas funcionam; em outros momentos, você ignora os detalhes até o fim do processo, porque isso implicaria a possibilidade de o projeto ser pensado como uma sequência de atividades. Na verdade, todas essas atividades são intercaladas e, assim, influenciam-se mutuamente.

Essas atividades de processo mencionadas são ilustradas com o projeto de parte do software para a estação meteorológica no deserto que apresentei no Capítulo 1. As estações meteorológicas no deserto são implantadas em áreas remotas. Cada estação meteorológica registra informações meteorológicas locais e periodicamente as transfere para um sistema geral de informações por meio de um link de satélite.

7.1.1 Contexto e interações do sistema

O primeiro estágio de qualquer processo de projeto de software é o desenvolvimento de uma compreensão dos relacionamentos entre o software que está sendo projetado e seu ambiente externo. Isso é essencial para decidir como oferecer a funcionalidade requerida para o sistema e como estruturar o sistema para se comunicar com seu ambiente. A compreensão do contexto também permite estabelecer os limites do sistema.

Definir os limites do sistema ajuda a decidir quais recursos serão implementados no sistema que está sendo projetado e quais recursos estão em outros sistemas associados. Nesse caso, você precisa decidir como a funcionalidade é distribuída entre o sistema de controle para todas as estações meteorológicas e o software embutido na estação meteorológica em si.

Modelos de contexto do sistema e modelos de interação apresentam visões complementares dos relacionamentos entre um sistema e seu ambiente:

1. Um modelo de contexto do sistema é um modelo estrutural, que demonstra os outros sistemas no ambiente do sistema a ser desenvolvido.
2. Um modelo de interação é um modelo dinâmico que mostra como o sistema interage com seu ambiente quando ativo.

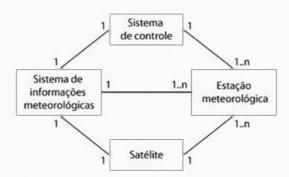
O modelo de contexto de um sistema pode ser representado por associações. Estas simplesmente mostram que existem alguns relacionamentos entre as entidades envolvidas na associação. Nesse momento, a natureza dos relacionamentos é especificada. Portanto, você pode documentar o ambiente do sistema com um diagrama de blocos simples, mostrando as entidades do sistema e de suas associações. Essa situação é ilustrada na Figura 7.1, que mostra que os sistemas no ambiente de cada estação meteorológica são um sistema de informações meteorológicas, um sistema de satélites a bordo e um sistema de controle. As informações de cardinalidade sobre o link mostram que há um sistema de controle com várias estações meteorológicas, um satélite e um sistema de informações climáticas gerais.

Quando você modela as interações de um sistema com seu ambiente, deve usar uma abordagem abstrata sem muitos detalhes. Uma maneira de fazer isso é usar um modelo de caso de uso. Como discutido nos capítulos 4 e 5, cada caso de uso representa uma interação com o sistema. Cada interação possível é nomeada em uma elipse, e a entidade externa envolvida na interação é representada por um boneco palito.

O modelo de caso de uso para a estação meteorológica está na Figura 7.2. Este mostra que a estação meteorológica interage com o sistema de informações meteorológicas para relatar dados meteorológicos e o *status* do hardware da estação. Outras interações são com um sistema de controle que pode emitir comandos específicos de uma estação do tempo. Como expliquei no Capítulo 5, um boneco palito é usado na UML para representar outros sistemas, bem como usuários humanos.

Cada um desses casos de uso deve ser descrito em linguagem natural estruturada. Isso ajuda os projetistas a identificarem objetos no sistema e oferece uma compreensão do que o sistema se destina a fazer. Eu uso um

Figura 7.1 Sistema de contexto para estação meteorológica

Figura 7.2 Casos de uso da estação meteorológica

formato-padrão para essa descrição, o qual identifica claramente as informações que são trocadas, como a interação é iniciada, e assim por diante. Isso é mostrado no Quadro 7.1, que descreve o caso de uso 'Relatar clima' a partir da Figura 7.2. Exemplos de alguns outros casos de uso estão disponíveis na Internet.

7.1.2 Projeto de arquitetura

Uma vez definidas as interações entre o sistema de software e o ambiente do sistema, você pode usar essa informação como base para projetar a arquitetura do sistema. Certamente, você precisa combinar essa informação com seu conhecimento geral dos princípios de projeto de arquitetura e com conhecimentos mais detalhados sobre o domínio. Você identifica os principais componentes do sistema e suas interações e, então, pode organizar os componentes usando um padrão de arquitetura, como um modelo em camadas ou cliente-servidor. No entanto, nesse estágio, isso não é essencial.

O projeto de arquitetura em alto nível para o software da estação meteorológica é mostrado na Figura 7.3. A estação meteorológica é composta de subsistemas independentes que se comunicam pela transmissão de mensagens em uma infraestrutura comum, apresentada na Figura 7.3 como 'Link de comunicação'. Cada subsistema escuta as mensagens sobre essa infraestrutura e pega as mensagens destinadas a eles. Esse é outro estilo de arquitetura comumente usado, além dos estilos já descritos no Capítulo 6.

Quadro 7.1 Descrição de caso de uso – Relatar clima

Caso de uso: Relatar clima

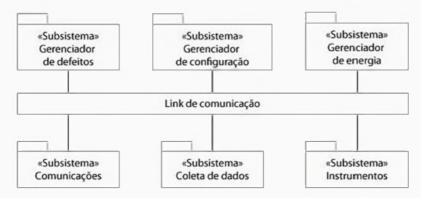
Atores: Sistema de informações meteorológicas, estação meteorológica

Dados: A estação meteorológica envia um resumo dos dados meteorológicos coletados a partir dos instrumentos, no período de coleta, para o sistema de informações meteorológicas. Os dados enviados são o máximo, mínimo e médio das temperaturas de solo e de ar; a máxima, mínima e média da pressão do ar; a velocidade máxima, mínima e média do vento; a precipitação de chuva total e a direção do vento, amostrados a cada cinco minutos.

Estímulo: O sistema de informações meteorológicas estabelece um link de comunicação via satélite com a estação e solicita a transmissão dos dados.

Resposta: Os dados resumidos são enviados para o sistema de informações meteorológicas.

Comentários: Geralmente, solicita-se que as estações meteorológicas enviem relatórios a cada hora, mas essa frequência pode diferir de uma estação para a outra e pode ser modificada no futuro.

Figura 7.3 Arquitetura de alto nível da estação meteorológica

Por exemplo, quando o subsistema de comunicações recebe um comando de controle, como o desligamento, o comando é capturado por cada um dos outros subsistemas, que se desligam de maneira correta. A principal vantagem dessa arquitetura é que é fácil suportar diferentes configurações de subsistemas, pois o remetente de uma mensagem não precisa endereçar a mensagem a um subsistema específico.

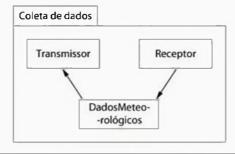
A Figura 7.4 mostra a arquitetura do subsistema de coleta de dados, incluída na Figura 7.3. Os objetos 'Transmissor' e 'Receptor' estão preocupados com o gerenciamento das comunicações, e o objeto 'Dados meteorológicos' encapsula a informação capturada a partir dos instrumentos e transmitida ao sistema de informações meteorológicas. Essa organização segue o modelo produtor-consumidor discutido no Capítulo 20.

7.1.3 Identificação dos objetos de classe

Nesse estágio do processo, você deve ter algumas ideias sobre os objetos essenciais para o sistema que está projetando. Enquanto sua compreensão do projeto se desenvolve, você refina suas ideias sobre os objetos do sistema. A descrição do caso de uso ajuda a identificar objetos e operações no sistema. A partir da descrição do caso de uso 'Relatar clima', é óbvio que os objetos que representam os instrumentos que coletam dados meteorológicos serão necessários, assim como um objeto representando o resumo desses dados. Geralmente, você também precisa de um objeto do sistema de alto nível ou objetos que sintetizem as interações do sistema definidas nos casos de uso. Com esses objetos em mente, você pode começar a identificar as classes de objeto no sistema.

Foram feitas várias propostas sobre como identificar as classes de objetos em sistemas orientados a objetos:

1. Usar uma análise gramatical de uma descrição em linguagem natural do sistema a ser construído. Objetos e atributos são substantivos, operações ou serviços são verbos (ABBOTT, 1983).
2. Usar entidades tangíveis (coisas) no domínio de aplicação, como aviões; papéis, como gerente ou médico; eventos, como pedidos; interações, como reuniões; locais, como escritórios; unidades organizacionais, como empresas, e assim por diante (COAD e YOURDON, 1990; SHLAER e MELLOR, 1988; WIRFS-BROCK et al., 1990).
3. Usar uma análise baseada em cenários, na qual diversos cenários de uso do sistema são identificados e analisados separadamente. À medida que cada cenário é analisado, a equipe responsável pela análise deve identificar os objetos necessários, atributos e operações (BECK e CUNNINGHAM, 1989).

Figura 7.4 Arquitetura do sistema de coleta de dados

Na prática, você precisa usar diversas fontes de conhecimento para descobrir as classes de objetos. As classes de objetos, atributos e operações, inicialmente identificados a partir da descrição informal do sistema, podem ser um ponto de partida para o projeto. A partir do conhecimento do domínio de aplicação ou da análise de cenário, mais informações podem ser usadas para refinar e estender os objetos iniciais. Essa informação pode ser coletada a partir de documentos de requisitos, conversas com usuários, ou a partir de análises dos sistemas existentes.

Na estação meteorológica no deserto, a identificação de objetos é baseada no hardware tangível no sistema. Eu não tenho espaço para incluir todos os objetos do sistema aqui, mas na Figura 7.5 mostro cinco classes de objetos. Os objetos 'Termômetro de chão', 'Anemômetro' e 'Barômetro' são objetos de domínio de aplicação, e os objetos 'EstaçãoMeteorológica' e 'DadosMeteorológicos' foram identificados a partir da descrição de sistema e da descrição de cenário (caso de uso):

1. A classe de objeto EstaçãoMeteorológica fornece a interface básica da estação meteorológica com seu ambiente. Suas operações refletem as interações mostradas no Quadro 7.1. Nesse caso, eu uso uma classe única de objeto para encapsular todas essas interações, mas em outros projetos você poderia projetar a interface do sistema como várias classes diferentes.
2. A classe de objeto DadosMeteorológicos é responsável por processar o comando 'Relatar clima'. Este envia os dados resumidos dos instrumentos da estação meteorológica para o sistema de informações meteorológicas.
3. As classes de objeto Termômetro de chão, Anemômetro e Barômetro estão diretamente relacionadas com os instrumentos do sistema. Estes refletem as entidades tangíveis de hardware no sistema, e as operações estão interessadas em controlar o hardware. Esses objetos funcionam de forma autônoma para coletar dados na frequência especificada e armazenam os dados coletados localmente. Quando requisitados, esses dados são entregues ao objeto DadosMeteorológicos.

Você usa o conhecimento do domínio da aplicação para identificar outros objetos, atributos e serviços. Sabemos que, muitas vezes, as estações meteorológicas estão em lugares remotos e incluem diversos instrumentos que, eventualmente, falham. Falhas do instrumento devem ser comunicadas automaticamente. Isso significa que você precisa de atributos e operações para verificar o correto funcionamento dos instrumentos. Existem muitas estações meteorológicas remotas, e cada uma deve ter seu próprio identificador.

Nesse estágio do processo de projeto, você deve concentrar-se nos próprios objetos, sem pensar em como eles podem ser implementados. Depois de identificar os objetos, você deve refinar o projeto do objeto. Você observa as características comuns e, em seguida, projeta a hierarquia de herança para o sistema. Por exemplo, você pode identificar uma superclasse 'Instrumento', que define as características comuns de todos os instrumentos, assim como um identificador, e obter operações e testar. Você também pode adicionar novos atributos e operações à superclasse, tal como um atributo que mantenha a frequência da coleta de dados.

Figura 7.5 Objetos da estação meteorológica

EstaçãoMeteorológica
Identificador
relatarClima ( ) relatarStatus( ) economizarEnergia (instrumentos) controlarRemoto (comandos) reconfigurar(comandos) reiniciar (instrumentos) desligar (instrumentos)

DadosMeteorológicos
temperaturaAr temperaturaChão velocidadeVento direçãoVento pressão precipitaçãoChuva
coletar ( ) resumir ( )

Termômetro de chão
get_Ident temperatura
obter( ) testar( )

Anemômetro
an_Ident velocidadeVento direçãoVento
obter ( ) testar ( )

Barômetro
bar_Ident pressão altura
obter ( ) testar ( )

7.1.4 Modelos de projeto

Modelos de projeto ou de sistema, como discuti no Capítulo 5, mostram os objetos ou classes de objetos em um sistema. Eles também mostram as associações e relacionamentos entre essas entidades. Esses modelos são a ponte entre os requisitos do sistema e a implementação de um sistema. Eles precisam ser abstratos, para que os detalhes desnecessários não escondam os relacionamentos entre eles e os requisitos do sistema. No entanto, também devem incluir detalhes suficientes para que os programadores possam tomar decisões de implementação.

Geralmente, você contorna esse tipo de conflito desenvolvendo modelos em diferentes níveis de detalhe. Sempre que houver links estreitos entre os engenheiros de requisitos, projetistas e programadores, os modelos abstratos podem ser tudo o que é necessário. Decisões específicas de projeto podem ser feitas enquanto o sistema é implementado, com problemas resolvidos por meio de discussões informais. Quando os links entre os especificadores do sistema, projetistas e programadores são indiretos (por exemplo, quando um sistema está sendo projetado em uma parte de uma organização, mas implementado em outros lugares), modelos mais pormenorizados poderão ser necessários.

Portanto, um passo importante no processo de projeto é decidir os modelos de projeto que você precisa e o nível de detalhes para esses modelos. Isso depende do tipo de sistema que está sendo desenvolvido. A maneira de projetar um sistema sequencial de processamento de dados é diferente da maneira de projetar um sistema embutido de tempo real; assim você terá modelos de projeto diferentes. A UML suporta 13 tipos diferentes de modelos, mas, como discutido no Capítulo 5, você raramente usa tudo isso. Minimizar o número de modelos produzidos reduz os custos do projeto e o tempo necessário para completar o processo. Quando você usa a UML para desenvolver um projeto, você normalmente desenvolve dois tipos de modelos de projeto:

1. Modelos estruturais, os quais descrevem a estrutura estática do sistema, usando as classes de objetos e seus relacionamentos. Relacionamentos importantes que podem ser documentados nesse estágio são os de generalização (herança), usa/usado por, e composição.
2. Modelos dinâmicos, os quais descrevem a estrutura dinâmica do sistema e mostram as interações entre os objetos do sistema. Interações que podem ser documentadas incluem a sequência de solicitações de serviço feitas pelos objetos e as mudanças de estado que são disparadas por essas interações de objetos.

Existem três modelos particularmente úteis para acrescentar detalhes aos modelos de caso de uso e de arquitetura, nos estágios iniciais do processo de projeto:

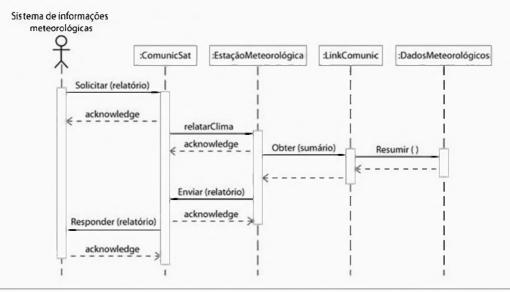
1. Modelos de subsistema, que mostram agrupamentos lógicos de objetos em subsistemas coerentes. Estes são representados por uma forma de diagrama de classe com cada subsistema exibido como um pacote de objetos. Modelos de subsistemas são modelos estáticos (estruturais).
2. Modelos de sequência, que mostram a sequência de interações de objetos. Estes são representados por um diagrama de sequência ou de colaboração da UML. Modelos de sequência são dinâmicos.
3. Modelos de máquina de estado, que mostram como objetos individuais mudam de estado em resposta aos eventos. Estes são representados na UML por diagramas de estado. Modelos de máquina de estados são modelos dinâmicos.

Um modelo de subsistema é um modelo estático útil, pois mostra como um projeto se organiza em grupos de objetos logicamente relacionados. Eu já mostrei esse tipo de modelo na Figura 7.3 para apresentar os subsistemas do sistema de mapeamento de tempo. Assim como modelos de subsistema, você também poderá projetar modelos de objetos detalhados, mostrando todos os objetos no sistema e suas associações (herança, generalização, agregação etc.). No entanto, existe um perigo em fazer muita modelagem. Você não deve tomar decisões pormenorizadas sobre a implementação, elas devem ser deixadas para os programadores de sistema.

Modelos de sequência são modelos dinâmicos que descrevem, para cada modo de interação, a sequência de interações de objetos ocorridas. Ao documentar um projeto, você deve produzir um modelo de sequência para cada interação significativa. Se você desenvolveu um modelo de caso de uso, então deve haver um modelo de sequência para cada caso de uso que você identificou.

A Figura 7.6 é um exemplo de modelo de sequência, mostrado como um diagrama de sequência UML. Esse diagrama mostra a sequência de interações que ocorrem quando um sistema externo envia um pedido dos dados resumidos da estação meteorológica. Os diagramas de sequência devem ser lidos de cima para baixo:

1. O objeto ComunicSat recebe uma solicitação do sistema de informações meteorológicas para coletar um relatório de clima de uma estação meteorológica. Ele acusa o recebimento da solicitação. A seta na mensagem enviada indica que o sistema externo não espera uma resposta, mas pode continuar com outro processamento.

Figura 7.6 Diagramas de sequência descrevendo coleta de dados

2. O ComunicSat envia uma mensagem para EstaçãoMeteorológica por meio de um link de satélite para criar um resumo dos dados meteorológicos coletados. Novamente, a seta indica que o ComunicSat não fica em suspenso esperando por uma resposta.
3. EstaçãoMeteorológica envia uma mensagem a um objeto LinkComunic para resumir os dados meteorológicos. Nesse caso, o estilo quadrado da seta indica que a instância da classe de objeto EstaçãoMeteorológica aguarda uma resposta.
4. LinkComunic chama o método resumir no objeto DadosMeteorológicos e aguarda uma resposta.
5. O resumo dos dados meteorológicos é calculado e retornado para EstaçãoMeteorológica por meio do objeto LinkComunic.
6. Em seguida, EstaçãoMeteorológica chama o objeto ComunicSat para transmitir os dados resumidos para o sistema de informações meteorológicas, por meio do sistema de comunicações por satélite.

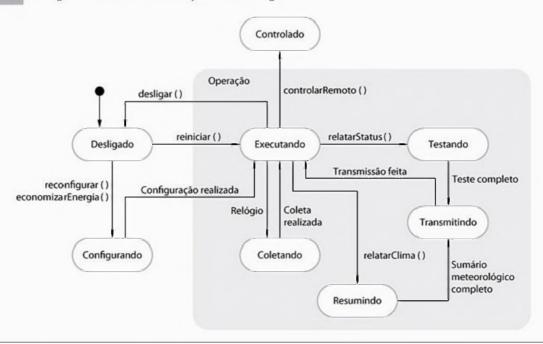
Os objetos ComunicSat e EstaçãoMeteorológica podem ser implementados como processos concorrentes, cuja execução pode ser suspensa e reiniciada. A instância do objeto ComunicSat escuta as mensagens do sistema externo, decodifica essas mensagens e inicia as operações da estação meteorológica.

Os diagramas de sequência são usados para modelar o comportamento combinado de um grupo de objetos, mas você também pode querer resumir o comportamento de um objeto ou de um subsistema em resposta às mensagens e aos eventos. Para fazer isso, você pode usar um modelo de máquina de estado que mostre como a instância do objeto muda de estado, dependendo das mensagens que recebe. A UML inclui diagramas de estado, que foram inventados por Harel (1987) para descrever modelos de máquina de estado.

A Figura 7.7 é um diagrama de estado para o sistema de estação meteorológica que mostra como ele responde a pedidos de vários serviços.

Você pode ler esse diagrama como segue:

1. Se o estado do sistema é Desligado, ele pode responder a uma mensagem reiniciar ( ), reconfigurar ( ) ou economizarEnergia ( ). A seta com uma bolha preta, não marcada, indica que o estado Desligado é o estado inicial. Uma mensagem reiniciar ( ) causa a transição para a operação normal. Tanto a mensagem de economiaEnergia ( ) como a de reconfigurar ( ) podem causar uma transição para um estado no qual o sistema se reconfigura. O diagrama de estado mostra que a reconfiguração só é permitida se o sistema esteve desligado.
2. No estado Executando, o sistema espera futuras mensagens. Se uma mensagem desligar ( ) é recebida, o objeto retorna para o estado Desligado.
3. Se uma mensagem relatarClima ( ) é recebida, o sistema se move para o estado Resumindo. Quando o resumo estiver concluído, o sistema passa para um estado Transmitindo, no qual as informações são transmitidas para o sistema remoto. Em seguida, retorna ao estado Executando.

Figura 7.7 Diagrama de estado da estação meteorológica

4. Se uma mensagem relatarStatus ( ) é recebida, o sistema se move para o estado Testando, depois para o estado Transmitindo, antes de retornar ao estado Executando.
5. Se um sinal do relógio é recebido, o sistema passa ao estado Coletando, em que coleta os dados dos instrumentos. Cada instrumento, por sua vez, é instruído para coletar os dados dos sensores associados.
6. Se uma mensagem controlarRemoto ( ) é recebida, o sistema se move para um estado Controlado, em que ele responde a um conjunto de diferentes mensagens da sala de controle remoto. Estas não são mostradas nesse diagrama.

Diagramas de estado são modelos úteis de alto nível de um sistema ou do funcionamento de um objeto. Geralmente, você não precisa de um diagrama de estado para todos os objetos do sistema. Muitos dos objetos em um sistema são relativamente simples, e um modelo de estado acrescenta detalhes desnecessários ao projeto.

7.1.5 Especificações de interface

Uma parte importante de qualquer processo de projeto é a especificação das interfaces entre os componentes do projeto. Você precisa especificar as interfaces de forma que os objetos e os subsistemas possam ser projetados em paralelo. Depois que uma interface tenha sido especificada, os desenvolvedores de outros objetos podem supor que ela será implementada.

O projeto de interface está preocupado com a especificação dos detalhes da interface para um objeto ou para um grupo de objetos. Isso significa definir as assinaturas e a semântica dos serviços fornecidos pelo objeto ou por um grupo de objetos. As interfaces podem ser especificadas em UML usando-se a mesma notação como um diagrama de classe. No entanto, não existe a seção de atributos, e o estereótipo UML <<interface>> deve ser incluído na parte do nome. A semântica da interface pode ser definida usando-se a linguagem de restrição de objetos (OCL, do inglês *object constraint language*). Explico isso no Capítulo 17, em que discuto a engenharia de software baseada em componentes. Também mostro uma forma alternativa de representar as interfaces em UML.

Você não deve incluir detalhes sobre a representação de dados em um projeto de interface, já que atributos não são definidos em uma especificação de interface. No entanto, você deve incluir as operações de acesso e atualização de dados. Como a representação de dados é oculta, ela pode ser facilmente alterada sem afetar os objetos que usam esses dados. O que gera um projeto que é inerentemente mais manutenível. Por exemplo, uma representação de uma pilha, via vetor, pode ser alterada para uma representação de lista sem afetar os outros objetos que usam a pilha. Pelo contrário, muitas vezes faz sentido expor os atributos em um modelo de projeto estático, pois essa é a forma mais compacta de ilustrar características essenciais dos objetos.

Não existe um relacionamento simples 1:1 entre objetos e interfaces. O mesmo objeto pode ter várias interfaces, cada uma com um ponto de vista sobre os métodos que ele fornece. Isso é suportado diretamente em Java, em que as interfaces são declaradas separadamente a partir de objetos e objetos 'implementam' interfaces. Da mesma maneira, um grupo de objetos pode ser acessado por meio de apenas uma interface.

A Figura 7.8 mostra duas interfaces que podem ser definidas para a estação meteorológica. A interface do lado esquerdo é uma interface de relatório que define os nomes das operações para gerar relatórios de clima e *status*. Estes mapeiam diretamente para as operações no objeto EstaçãoMeteorológica. A interface de controle remoto fornece quatro operações, que mapeiam para um único método no objeto EstaçãoMeteorológica. Nesse caso, as operações individuais são codificadas no *string* de comando associado com o método controlarRemoto, na Figura 7.5.

7.2 Padrões de projeto

Os padrões de projeto foram obtidos a partir das ideias apresentadas por Christopher Alexander (ALEXANDER et al., 1977), que sugeriu haver padrões comuns de projeto de prédios que eram inerentemente agradáveis e eficazes. O padrão é uma descrição do problema e da essência de sua solução, de modo que a solução possa ser reusada em diferentes contextos. O padrão não é uma especificação detalhada. Em vez disso, você pode pensar nele como uma descrição de conhecimento e experiência, uma solução já aprovada para um problema comum.

Uma citação do site da Hillside Group (<http://hillside.net>), dedicado a manter informações sobre os padrões, sintetiza seu papel no reúso:

> *Padrões e Linguagens de Padrões são formas de descrever as melhores práticas, bons projetos e capturar a experiência de uma forma que torne possível a outros reusar essa experiência.*

Os padrões tiveram um enorme impacto no projeto de software orientado a objetos. Além de serem soluções já testadas para problemas comuns, tornaram-se um vocabulário para falar sobre um projeto. Você pode, portanto, explicar seu projeto por meio de descrições dos padrões que você usou. Isso é particularmente verdadeiro para os padrões de projeto mais conhecidos que foram originalmente descritos pela 'Gangue dos Quatro' em seu livro de padrões (GAMMA et al., 1995). Outras descrições de padrões particularmente importantes são as publicadas em uma série de livros de autores da Siemens, uma grande empresa europeia de tecnologia (BUSCHMANN et al., 1996;. BUSCHMANN et al., 2007a; BUSCHMANN et al., 2007b; KIRCHER e JAIN, 2004; SCHMIDT et al., 2000).

Os padrões de projeto são normalmente associados com projeto orientado a objetos. Muitas vezes, os padrões publicados contam com as características de objetos, como herança e polimorfismo para fornecer generalidade. No entanto, o princípio geral de encapsular a experiência em um padrão é aquele igualmente aplicável a qualquer tipo de projeto de software. Então, você poderia ter padrões de configuração para os sistemas COTS. Os padrões são uma maneira de reusar o conhecimento e a experiência de outros projetistas.

Os quatro elementos essenciais dos padrões de projeto foram definidos pela 'Gangue dos Quatro', em seu livro de padrões:

1. Um nome que seja uma referência significativa para o padrão.
2. Uma descrição da área de problema que explique quando o modelo pode ser aplicado.
3. A descrição da solução das partes da solução de projeto, seus relacionamentos e suas responsabilidades. Essa não é uma descrição do projeto concreto; é um modelo para uma solução de projeto que pode ser instanciado de diferentes maneiras. Costuma ser expresso graficamente e mostra os relacionamentos entre os objetos e suas classes na solução.

Figura 7.8 Interfaces da estação meteorológica

«Interface»
Controle remoto
iniciarInstrumento (instrumento): iStatus
pararInstrumento (instrumento): iStatus
coletarDados (instrumento): iStatus
fornecerDados (instrumento): string

4. Uma declaração das consequências — os resultados e compromissos — da aplicação do padrão. Pode ajudar os projetistas a entenderem quando um padrão pode ou não ser usado em uma situação particular.

Gamma et al. (1995) quebram a descrição do problema em motivação (uma descrição do motivo pelo qual o padrão é útil) e aplicabilidade (uma descrição das situações nas quais o padrão pode ser usado). Sob a descrição da solução, eles descrevem a estrutura do padrão, participantes, colaborações e implementação.

Para ilustrar a descrição do padrão, eu uso o padrão Observer, extraído do livro de Gamma et al. (1995), que é mostrado no Quadro 7.2. Em minha descrição, uso os quatro elementos descritivos essenciais e incluo uma breve declaração do que o padrão pode fazer. Esse padrão pode ser usado em situações nas quais as diferentes apresentações do estado do objeto são requeridas. Ele separa o objeto que deve ser exibido a partir de diferentes formas de apresentação. Essa situação é ilustrada na Figura 7.9, que mostra duas representações gráficas do mesmo conjunto de dados.

As representações gráficas são normalmente usadas para ilustrar as classes de objeto em padrões e seus relacionamentos. Estes suplementam a descrição do padrão e acrescentam detalhes à descrição da solução. A Figura 7.10 é a representação do padrão Observer em UML.

Para usar padrões em seu projeto, você precisa reconhecer que qualquer problema de projeto que você esteja enfrentando pode ter um padrão associado que pode ser aplicado. Exemplos desses problemas, documentados no livro original de padrões da 'Gangue dos Quatro', incluem:

1. Dizer a vários objetos que o estado de algum outro objeto mudou (padrão Observer).
2. Arrumar as interfaces para vários objetos relacionados que, muitas vezes, foram desenvolvidos de forma incremental (padrão Façade).
3. Fornecer uma forma padronizada de acesso aos elementos de uma coleção, independentemente de como essa coleção é implementada (padrão Iterator).
4. Permitir a possibilidade de estender a funcionalidade de uma classe existente em *run-time* (padrão Decorator).

Os padrões suportam reúso de conceito em alto nível. Ao tentar reusar componentes executáveis, você inevitavelmente é limitado pelas decisões de projeto detalhado feitas pelos implementadores desses componentes. Elas vão desde os algoritmos particulares que têm sido usados para implementar os componentes até os objetos e os tipos na interface de componentes. Quando essas decisões de projeto entram em conflito com

Quadro 7.2 O padrão Observer

Nome do padrão:	Observer
Descrição:	Separa o *display* do estado de um objeto a partir do objeto em si e permite que sejam fornecidos *displays* alternativos. Quando o estado do objeto muda, todos os *displays* são automaticamente notificados e atualizados para refletir a mudança.
Descrição do problema:	Em muitas situações, você precisa fornecer vários *displays* de informações do estado, como um *display* gráfico e em tabela. Nem todos eles podem ser conhecidos quando a informação é especificada. Todas as apresentações alternativas devem apoiar a interação e, quando o estado é alterado, todos os *displays* devem ser atualizados. Esse padrão pode ser usado em todas as situações em que mais de um formato de *display* de informações de estado é necessário, e em que saber sobre os formatos de *display* específicos usados não é necessário para o objeto que mantém as informações do estado.
Descrição da solução:	Trata-se de dois objetos abstratos, Subject e Observer, e dois objetos concretos, ConcreteSubject e ConcreteObject, que herdam os atributos dos objetos abstratos relacionados. Os objetos abstratos incluem as operações gerais aplicáveis em todas as situações. O estado a ser exibido é mantido no ConcreteSubject, que herda as operações de Subject permitindo adicionar ou remover Observers (cada Observer corresponde a um *display*) e emitir uma notificação quando o estado mudar. O ConcreteObserver mantém uma cópia do estado do ConcreteSubject e implementa a interface atualizar ( ) do Observer, que permite que essas cópias sejam mantidas nessa etapa. Automaticamente, o ConcreteObserver exibe o estado e reflete as mudanças sempre que o estado é atualizado. O modelo UML do padrão é mostrado na Figura 7.10.
Consequências:	O Subject só conhece o Observer abstrato e não sabe detalhes da classe concreta. Portanto, há um acoplamento mínimo entre esses objetos. Devido a essa falta de conhecimento, as otimizações que melhoram o desempenho do *display* são impraticáveis. As alterações no Subject podem causar uma série de atualizações ligadas aos Observers relacionados para serem geradas, algumas das quais podem não ser necessárias.

Figura 7.9 Múltiplos *displays*

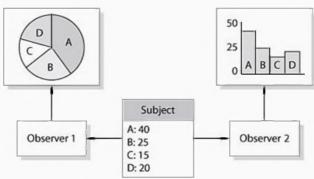
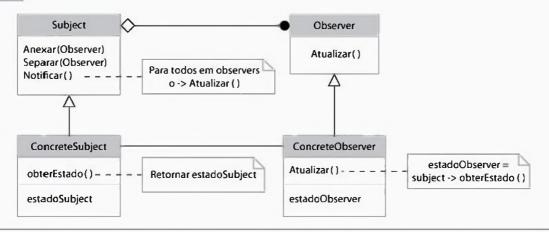



Figura 7.10 Um modelo UML do padrão Observer

seus requisitos específicos, o reúso de componentes é impossível ou apresenta ineficiências em seu sistema. O uso de padrões significa que você reusa as ideias, mas pode adaptar a aplicação para se adequar ao sistema que está desenvolvendo.

Quando você começa a projetar um sistema, pode ser difícil saber, antecipadamente, se vai precisar de determinado padrão. Portanto, muitas vezes, o uso de padrões em um processo envolve o desenvolvimento de um projeto, a experimentação com algum problema e, em seguida, o reconhecimento de que um padrão pode ser usado. Isso certamente é possível se você se concentrar nos 23 padrões de uso geral documentados no livro original de padrões. No entanto, se o problema for diferente, você pode ter dificuldade para encontrar um padrão adequado entre as centenas de padrões diferentes que têm sido propostos.

Os padrões são uma ótima ideia, mas, para usá-los efetivamente, você precisa de experiência de desenvolvimento de software. Você precisa reconhecer as situações em que um padrão pode ser aplicado. Programadores inexperientes, mesmo que tenham lido o livro de padrões, sempre acharão difícil decidir se podem reusar um padrão ou se é necessário desenvolver uma solução especial.

7.3 Questões de implementação

A engenharia de software inclui todas as atividades envolvidas no desenvolvimento de software, desde os requisitos iniciais do sistema até a manutenção e o gerenciamento do sistema implantado. O estágio mais crítico desse processo é, naturalmente, a implementação do sistema, estágio em que você cria uma versão executável do software. A implementação pode envolver o desenvolvimento de programas em alto ou baixo nível de linguagens de programação, bem como customização e adaptação de sistemas genéricos de prateleira, para atender aos requisitos específicos de uma organização.

Presumo que a maioria dos leitores deste livro entenderá os princípios de programação e terá alguma experiência na área. Como este capítulo se destina a oferecer uma abordagem independente de linguagem, não me concentrei em questões de boa prática da programação, pois elas precisam usar exemplos específicos de linguagem. Em vez disso, apresento alguns aspectos de implementação particularmente importantes para a engenharia de software e que, muitas vezes, não são cobertos em textos de programação. São eles:

1. *Reúso.* Os softwares mais modernos são construídos por meio do reúso de componentes existentes ou sistemas. Quando você está desenvolvendo um software, deve fazer o maior uso possível dos códigos existentes.
2. *Gerenciamento de configuração.* Durante o processo de desenvolvimento, são criadas muitas versões diferentes de cada componente de software. Se você não acompanhar essas versões em um sistema de gerenciamento de configuração, estará suscetível a incluir as versões erradas desses componentes em seu sistema.
3. *Desenvolvimento* host-target. A produção de software não costuma executar no mesmo computador como no ambiente de desenvolvimento de software. Em vez disso, você desenvolve em um computador (o sistema *host*) e executa em outro (o sistema *target*). Os sistemas *host* e *target* são, por vezes, do mesmo tipo, mas, muitas vezes, completamente diferentes.

7.3.1 Reúso

De 1960 a 1990, o software mais novo foi desenvolvido a partir do zero. Todos os códigos foram escritos em linguagem de programação de alto nível. O único reúso significativo de software foi o de funções e objetos em bibliotecas de linguagem de programação. No entanto, os custos e a pressão de cronograma significavam que essa abordagem se tornava cada vez menos viável, especialmente para sistemas comerciais e baseados na Internet. Consequentemente, surgiu uma abordagem de desenvolvimento baseada no reúso de softwares existentes. Atualmente, esta é usada com frequência para sistemas de negócios, softwares científicos e, cada vez mais, em engenharia de sistemas embutidos.

O reúso de software é possível em vários níveis diferentes:

1. *O nível de abstração.* Nesse nível, você não reúsa o software diretamente, mas usa o conhecimento das abstrações de sucesso no projeto de seu software. Os padrões de projeto e de arquitetura (abordados no Capítulo 6) são formas de representar o conhecimento abstrato para reúso.
2. *O nível de objeto.* Nesse nível, você reúsa objetos diretamente de uma biblioteca em vez de escrever um código. Para implementar esse tipo de reúso, você tem de encontrar bibliotecas adequadas e descobrir se os objetos e métodos oferecem a funcionalidade que você precisa. Por exemplo, se você precisa processar mensagens de correio em um programa Java, você pode usar objetos e métodos de uma biblioteca JavaMail.
3. *O nível de componentes.* Componentes são coleções de objetos e classes de objetos que funcionam em conjunto para fornecer funções e serviços relacionados. Muitas vezes, você tem de se adaptar e ampliar o componente adicionando um código próprio. Um exemplo de reúso em nível de componente é aquele no qual você constrói sua interface de usuário usando um *framework*. Este é um conjunto de classes de objetos em geral que implementam manipulação de eventos, gerencimento de *displays* etc. Você adiciona as conexões com os dados a serem exibidos e escreve o código para definir detalhes específicos do *display*, como o *layout* da tela e as cores.
4. *O nível de sistema.* Nesse nível, você reúsa os sistemas de aplicação inteiros, o que geralmente envolve algum tipo de configuração desses sistemas. Essas configurações podem ser feitas por meio da adição e modificação do código (se você estiver reusando uma linha de produtos de software) ou pelo uso de interface de configuração do próprio sistema. A maioria dos sistemas comerciais é criada dessa forma, em que sistemas genéricos de COTS (*commercial off-the-shelf*) são adaptados e reusados. Às vezes, essa abordagem pode envolver o reúso e a integração de diversos sistemas para criar um novo.

Ao reusar softwares existentes, você pode desenvolver novos sistemas mais rapidamente, com menos riscos de desenvolvimento e custos mais baixos. Como o software reusado foi testado em outras aplicações, deve ser mais confiável que o novo software. No entanto, existem custos associados ao reúso:

1. Os custos de tempo gasto na procura do software para reúso e na avaliação sobre ele atender ou não às necessidades. Talvez você precise testar o software para ter certeza de que vai funcionar em seu ambiente, sobretudo se ele for diferente do ambiente de desenvolvimento.
2. Quando se aplicam os custos de aquisição do software reusável. Para grandes sistemas de prateleira, esses custos podem ser muito elevados.

3. Os custos de adaptação e configuração dos componentes de software reusável ou sistemas para refletir os requisitos do sistema que você está desenvolvendo.
4. Os custos de integração de componentes de software reusável (se você estiver usando software de diferentes fontes) com o novo código que você desenvolveu. A integração de softwares reusáveis de diferentes fornecedores pode ser difícil e cara, pois os fornecedores podem fazer suposições conflitantes sobre como seus respectivos softwares serão reusados.

Como reusar o conhecimento e o software existente deve ser a primeira coisa em que você deve pensar ao iniciar um projeto de desenvolvimento de software. Você deve considerar as possibilidades de reúso antes de projetar o software em detalhes, assim como pode querer adaptar seu projeto para reusar ativos de software existentes. Como discutido no Capítulo 2, em um processo de desenvolvimento orientado ao reúso, você procura elementos reusáveis e, em seguida, altera seu projeto e requisitos para fazer melhor uso deles.

Para um grande número de sistemas de aplicações, a engenharia de software realmente significa o reúso de software. Por isso, eu dedico vários capítulos, na seção de tecnologias de software deste livro, a esse tópico (capítulos 16, 17 e 19).

7.3.2 Gerenciamento de configuração

No desenvolvimento de software a mudança acontece o tempo todo, de modo que o gerenciamento de mudanças é absolutamente essencial. Quando uma equipe de pessoas está desenvolvendo um software, é necessário assegurar que os membros da equipe não interfiram no trabalho uns dos outros. Ou seja, se duas pessoas estão trabalhando em um componente, as mudanças precisam ser coordenadas. Caso contrário, um programador pode fazer mudanças e escrever sobre o trabalho do outro. Você também precisa garantir que todos possam acessar as versões mais atualizadas dos componentes de software; caso contrário, os desenvolvedores podem refazer o trabalho que já foi feito. Quando algo dá errado com uma nova versão de um sistema, você precisa ser capaz de voltar para uma versão de trabalho do sistema ou componente.

Gerenciamento de configuração é o nome do processo geral de gerenciamento de um sistema de software em mudança. O objetivo do gerenciamento de configuração é apoiar o processo de integração do sistema para que todos os desenvolvedores possam acessar o código do projeto e os documentos relacionados de forma controlada, descobrir quais mudanças foram feitas, bem como compilar e ligar componentes para criar um sistema. Há, portanto, três atividades fundamentais no gerenciamento de configuração:

1. *Gerenciamento de versões*, em que o suporte é fornecido para manter o controle das diferentes versões de componentes de software. Sistemas de gerenciamento de versões incluem recursos para coordenar o desenvolvimento de diversos programadores. Eles bloqueiam um desenvolvedor que esteja escrevendo sobre um código submetido ao sistema por outra pessoa.
2. *Integração de sistemas*, em que o suporte é fornecido para ajudar os desenvolvedores a definir quais versões dos componentes são usadas para criar cada versão de um sistema. Essa descrição é, então, usada para construir um sistema automaticamente, compilando e ligando os componentes necessários.
3. *Rastreamento de problemas*, em que o suporte é fornecido para permitir aos usuários reportar *bugs* e outros problemas, além de permitir que todos os desenvolvedores possam ver quem está trabalhando nesses problemas e quando eles são resolvidos.

Ferramentas de gerenciamento de configuração de software apoiam cada uma das atividades mencionadas. Essas ferramentas podem ser projetadas para trabalharem juntas em um sistema abrangente de gerenciamento de mudanças, como o ClearCase (BELLAGIO e MILLIGAN, 2005). Nos sistemas integrados de gerenciamento de configuração, gerenciamento de versões, integração de sistemas e ferramentas de monitoramento de problemas, são projetadas em conjunto. Elas compartilham um estilo de interface de usuário e são integradas por meio de um repositório de código comum.

As ferramentas separadas, instaladas em um ambiente de desenvolvimento integrado, podem ser uma alternativa. O gerenciamento de versões pode ser apoiado por meio de um sistema de gerenciamento de versões tal qual o Subversion (PILATO et al., 2008), capaz de suportar desenvolvimento de multiequipes e multissites. O sistema de apoio à integração pode ser incorporado à linguagem ou depender de um conjunto de ferramentas separadas, como o sistema de construção GNU. Isso inclui aquela que talvez seja a ferramenta de integração mais conhecida, Unix Make. Sistemas de rastreamento de *bugs* ou de problemas, como Bugzilla, são usados para relatar *bugs* e outros problemas, bem como para controlar se estes foram corrigidos ou não.

Devido a sua importância na engenharia de software profissional, eu discuto gerenciamento de mudaças e de configuração em detalhes no Capítulo 25.

7.3.3 Desenvolvimento *host-target*

A maioria dos softwares de desenvolvimento é baseada em um modelo *host-target*. O software é desenvolvido em um computador (*host*), mas é executado em outra máquina (*target*). Mais genericamente, podemos falar de uma plataforma de desenvolvimento e uma de execução. Uma plataforma é mais do que apenas hardware. Ela inclui o sistema operacional instalado, além de softwares de apoio, como um sistema de gerenciamento de banco de dados ou, para plataformas de desenvolvimento, um ambiente de desenvolvimento interativo.

Às vezes, as plataformas de desenvolvimento e de execução são as mesmas, tornando possível desenvolver e testar o software na mesma máquina. Porém, é mais comum que sejam diferentes, de modo que você precise mover o software desenvolvido para a plataforma de execução para testes ou executar um simulador em sua máquina de desenvolvimento.

No desenvolvimento de sistemas embutidos, frequentemente se usam simuladores. Você simula dispositivos de hardware, como sensores, e os eventos no ambiente em que o sistema será implantado. Os simuladores aceleraram o processo de desenvolvimento de sistemas embutidos, assim como cada desenvolvedor pode ter sua própria plataforma de execução, sem necessidade de baixar o software para o hardware *target*. No entanto, os simuladores são caros para serem desenvolvidos e, dessa forma, só estão disponíveis para as arquiteturas mais populares de hardware.

Se o sistema *target* tiver instalado o *middleware* ou outro software que você precise usar, então você terá de ser capaz de testar o sistema com esse software. Pode ser pouco prático instalar esse software em sua máquina de desenvolvimento, mesmo que seja a mesma que a plataforma *target*, por causa das restrições de licença. Nessas circunstâncias, para testar o sistema você precisa transferir o código desenvolvido para a plataforma de execução.

Uma plataforma de desenvolvimento de software deve oferecer uma gama de ferramentas para apoiar os processos de engenharia de software. Estas podem incluir:

1. Um sistema compilador integrado e editor dirigido a sintaxe, que permita criar, editar e compilar o código.
2. Um sistema de depuração de linguagem.
3. Ferramentas de edição gráfica, como ferramentas para editar modelos da UML.
4. Ferramentas de teste, como JUnit (MASSOL, 2003), que podem executar, automaticamente, um conjunto de testes sobre uma nova versão de um programa.
5. Ferramentas de apoio ao projeto, que ajudem a organizar o código para diferentes projetos de desenvolvimento.

Assim como essas ferramentas padronizadas, seu sistema de desenvolvimento pode incluir ferramentas mais especializadas, como analisadores estáticos (discutidos no Capítulo 15). Normalmente, ambientes de desenvolvimento para equipes também incluem um servidor compartilhado que executa uma mudança e um sistema de gerenciamento de configuração, bem como, talvez, um sistema de apoio ao gerenciamento de requisitos.

As ferramentas de desenvolvimento de software são frequentemente agrupadas para criar um ambiente de desenvolvimento integrado (IDE). Um IDE é um conjunto de ferramentas de software que suportam diversos aspectos do desenvolvimento de software, dentro de um *framework* comum e uma interface de usuário. Geralmente, os IDEs são criados para apoiar o desenvolvimento de uma linguagem de programação específica, como Java. A linguagem IDE pode ser desenvolvida especialmente ou pode ser uma instância de um IDE de propósito geral, com ferramentas específicas de suporte de linguagem.

Um IDE de propósito geral é uma estrutura para hospedar ferramentas de software que fornece recursos de gerenciamento de dados para o software em desenvolvimento e mecanismos de integração que permitem às ferramentas que trabalhem juntas. O mais conhecido IDE de uso geral é o ambiente Eclipse (CARLSON, 2005). Esse ambiente é baseado em uma arquitetura *plug-in* para que possa ser especializada em diferentes linguagens e domínios de aplicação (CLAYBERG e RUBEL, 2006). Portanto, você pode instalar o Eclipse e adaptá-lo a suas necessidades específicas, adicionando *plug-ins*. Por exemplo, você pode adicionar um conjunto de *plug-ins* de apoio ao desenvolvimento de sistemas em rede em Java ou engenharia de sistemas embutidos usando C.

Como parte do processo de desenvolvimento, é preciso tomar decisões sobre como o software desenvolvido será implantado na plataforma *target*. Em sistemas embutidos, em que o *target* é geralmente um único computador, esse processo é simples. No entanto, para sistemas distribuídos, você precisa decidir sobre as plataformas específicas onde os componentes serão implantados. Os problemas que devem ser considerados ao se tomar essa decisão são:

1. *Os requisitos de hardware e software de um componente.* Se um componente é projetado para uma arquitetura de hardware específica ou depende de algum outro software do sistema, deve, certamente, ser implantado em uma plataforma que forneça o hardware requerido e o suporte de software.
2. *Os requisitos de disponibilidade de sistema.* Sistemas de alta disponibilidade podem exigir componentes a serem implantados em mais de uma plataforma. Isso significa que, em caso de falha da plataforma, uma implementação alternativa do componente estará disponível.
3. *Comunicações de componente.* Se houver um elevado nível de tráfego de comunicações entre os componentes, faz sentido implantá-los na mesma plataforma ou em plataformas que estejam fisicamente próximas. Isso reduz a latência de comunicações, o atraso entre o momento em que uma mensagem é enviada por um componente e recebida por outro.

Você pode documentar suas decisões em hardware e implantação de software usando diagramas de implantação da UML, que mostram como os componentes de software são distribuídos em diferentes plataformas de hardware.

Se você está desenvolvendo um sistema embutido, deve levar em conta as características do *target*, como seu tamanho físico, capacidades de energia, a necessidade de respostas em tempo real aos eventos de sensores, as características físicas dos atuadores e seu sistema operacional de tempo real. No Capítulo 20, discuto a engenharia de sistemas embutidos.

7.4 Desenvolvimento *open source*

O desenvolvimento *open source* é uma abordagem de desenvolvimento de software em que o código-fonte de um sistema de software é publicado e voluntários são convidados a participar no processo de desenvolvimento (RAYMOND, 2001). Suas raízes estão no Free Software Foundation (<http://www.fsf.org>), que defende que o código-fonte não deve ser proprietário, mas deve estar sempre disponível para os usuários analisarem e modificarem como quiserem. Havia uma suposição de que o código poderia ser controlado e desenvolvido por um pequeno grupo central, em vez de ser desenvolvido por usuários do código.

Os softwares *open source* estenderam essa ideia, usando a Internet para recrutar uma população muito maior de desenvolvedores voluntários. Muitos deles também são usuários do código. Pelo menos em princípio, qualquer contribuinte para um projeto *open source* pode relatar e corrigir *bugs* e propor novas características e funcionalidade. No entanto, na prática, sistemas *open source* de sucesso ainda contam com um grupo de desenvolvedores que controlam as mudanças no software.

O produto *open source* mais conhecido é, naturalmente, o sistema operacional Linux, amplamente usado como um sistema de servidor e, cada vez mais, como um ambiente de *desktop*. Outros importantes produtos *open source* são o Java, o servidor Web Apache e o sistema de gerenciamento de banco de dados mySQL. Os principais competidores da indústria da computação, como a IBM e a Sun, apoiam o movimento *open source* e baseiam seus produtos em software desse tipo. Existem milhares de outros sistemas e componentes *open source* menos conhecidos que também podem ser usados.

A aquisição de software *open source* costuma ser bastante barata ou gratuita, pois geralmente é possível baixar esses softwares sem custos. No entanto, se você precisar de documentação e suporte, você pode ter de pagar por isso, embora os custos sejam usualmente bastante baixos. O outro benefício-chave do uso de produtos *open source* é que sistemas *open source* maduros geralmente são muito confiáveis. A razão para isso é que eles têm uma grande população de usuários dispostos a corrigir os problemas em vez de os reportar ao desenvolvedor e esperar por novo *release* do sistema. Os *bugs* são descobertos e reparados mais rapidamente do que é possível, em geral, com softwares proprietários.

Para uma empresa envolvida no desenvolvimento de software, há duas questões de *open source* que devem ser consideradas:

1. O produto que está sendo desenvolvido deve fazer uso de componentes *open source*?
2. Uma abordagem *open source* deve ser usada para o desenvolvimento de software?

As respostas a essas perguntas dependem do tipo de software que está sendo desenvolvido, bem como dos antecedentes e da experiência da equipe de desenvolvimento.

Se você estiver desenvolvendo um produto de software para a venda, *time to market* e redução de custos são críticos. Se você estiver desenvolvendo em um domínio no qual existem sistemas *open source* de alta qualidade disponíveis, você pode economizar tempo e dinheiro usando esses sistemas. No entanto, se você estiver desenvolvendo software para um conjunto específico de requisitos organizacionais, usar componentes *open source* pode não ser uma opção. Você pode ter de integrar seu software com sistemas existentes que são incompatíveis com os sistemas *open source* disponíveis. Mesmo assim, pode ser mais rápido e mais barato modificar o sistema *open source* do que desenvolver novamente a funcionalidade que você precisa.

Mais e mais empresas de produtos estão usando uma abordagem *open source* para o desenvolvimento. Seu modelo de negócios não depende da venda de um produto de software, mas da venda de suporte para esse produto. Acredita-se que o envolvimento da comunidade *open source* permitirá o desenvolvimento de software de forma mais barata e mais rápida e criará uma comunidade de usuários para o software. Porém, repito, isso só é realmente aplicável para produtos gerais de software, e não para aplicações específicas da organização.

Muitas empresas acreditam que a adoção de uma abordagem *open source* vai revelar informações confidenciais de negócios a seus concorrentes, e por isso são relutantes em adotar esse modelo de desenvolvimento. No entanto, se você estiver trabalhando em uma pequena empresa, abrir o código-fonte de seu software poderá tranquilizar os clientes, que serão capazes de apoiar o software caso sua empresa saia do negócio.

Publicar o código-fonte de um sistema não significa que as pessoas da comunidade em geral, necessariamente, ajudarão com seu desenvolvimento. Os produtos *open source* mais bem-sucedidos têm sido os produtos de plataforma, e não os sistemas de aplicação. Há um número limitado de desenvolvedores que possam estar interessados em sistemas de aplicação especializada. Assim, fazer um sistema de software *open source* não garante o envolvimento da comunidade.

7.4.1 Licenças *open source*

Apesar de o livre acesso ao código-fonte ser um princípio fundamental do desenvolvimento *open source*, isso não significa que qualquer um pode fazer o que quiser com esse código. Legalmente, o desenvolvedor do código (seja uma empresa ou um indivíduo) ainda é seu proprietário. Desse modo, em uma licença de software *open source*, o proprietário pode colocar restrições em como o código é usado, incluindo as condições vinculadas legalmente (St. LAURENT, 2004). Alguns desenvolvedores *open source* acreditam que, se um componente *open source* é usado para desenvolver um novo sistema, esse sistema também deve ser *open source*. Outros estão dispostos a permitir que seu código seja usado sem essa restrição. Os sistemas desenvolvidos podem ser proprietários e vendidos como sistemas de código fechado.

A maioria das licenças *open source* derivam de um dos três modelos gerais:

1. A GNU General Public License (GPL). Essa é a chamada licença 'recíproca'; de forma simplista, significa que se você usar um software *open source* que esteja licenciado sob a licença GPL, você deve fazer um software *open source*.
2. A GNU Lesser General Public License (LGPL). Essa é uma variação da licença GPL, segundo a qual você pode escrever componentes que se ligam com código *open source* sem publicar a fonte desses componentes. No entanto, se você alterar o componente licenciado, você deve publicá-lo como *open source*.
3. A Berkley Standard Distribution (BSD) License. Essa é uma licença não recíproca, o que significa que você não é obrigado a republicar quaisquer alterações ou modificações feitas no código *open source*. Você pode incluir o código em sistemas proprietários que sejam vendidos. Se você usar componentes *open source*, deve reconhecer o criador original do código.

Questões de licenciamento são importantes porque, se você usar o software *open source* como parte de um produto de software, você pode ser obrigado pelos termos da licença a fazer seu próprio produto como *open source*. Se você está tentando vender seu software, você pode querer mantê-lo em segredo. Isso significa que, em seu desenvolvimento, você pode querer evitar o uso de software *open source* licenciado sob GPL.

Se você está construindo um software que roda em uma plataforma *open source*, como o Linux, as licenças não são um problema. No entanto, logo que você começa a incluir componentes *open source* em seu software, é necessário definir os processos e bancos de dados para manter o controle do que está sendo usado e suas condições da licença. Bayersdorfer (2007) sugere que as empresas de gerenciamento de projetos que usam código *open source* devem:

1. Estabelecer um sistema para manter informações sobre os componentes *open source* que são baixados e usados. É preciso manter uma cópia da licença para cada componente que era válido no momento em que foi usado. As licenças podem mudar, de modo que você precisa saber as condições com as quais concordou.
2. Estar ciente dos diferentes tipos de licenças e compreender como um componente é licenciado antes de ser usado. Você pode decidir usar um componente em um sistema, mas não em outro, porque você pretende usar esses sistemas de diferentes maneiras.
3. Estar ciente dos caminhos da evolução para os componentes. Você precisa saber um pouco sobre o projeto *open source* no qual os componentes são desenvolvidos para compreender como eles podem mudar no futuro.
4. Educar as pessoas sobre *open source*. Para assegurar o cumprimento das condições da licença não é suficiente ter os procedimentos em dia; você também precisa educar os desenvolvedores sobre código e licenças *open source*.
5. Ter os sistemas de auditoria em vigor. Os desenvolvedores com prazos apertados podem ser tentados a quebrar os termos de uma licença. Se possível, você deve ter um software para detectar e encerrar esse procedimento.
6. Participar da comunidade *open source*. Se você confia em produtos *open source*, deve participar da comunidade e ajudar a apoiar seu desenvolvimento.

O modelo de negócio de software está mudando. É cada vez mais difícil construir um negócio com a venda de sistemas de software especializados. Muitas empresas preferem fazer seu software *open source* e depois vender suporte e consultoria para os usuários do software. Isso tende a crescer com o uso cada vez maior de software *open source* e de softwares disponíveis.

PONTOS IMPORTANTES

- O projeto e a implementação de software são atividades intercaladas. O nível de detalhamento no projeto depende do tipo de sistema a ser desenvolvido e se está sendo usada uma abordagem ágil ou dirigida a planos.
- O processo de projeto orientado a objetos inclui atividades para projetar a arquitetura do sistema, identificar objetos no sistema, descrever o projeto usando diferentes modelos de objetos e documentar as interfaces dos componentes.
- Vários modelos diferentes podem ser produzidos durante um processo de projeto orientado a objetos. Estes incluem modelos estáticos (modelos de classes, modelos de generalização, modelos de associação) e modelos dinâmicos (modelos de sequência, modelos da máquina de estado).
- As interfaces de componentes devem ser definidas precisamente, de forma que outros objetos possam usá-las. Um estereótipo de interface da UML pode ser usado para definir as interfaces.
- Ao desenvolver um software, você sempre deve considerar a possibilidade de reusar um software existente, como seus componentes, serviços ou sistemas completos.
- O gerenciamento de configuração é o processo de gerenciamento de mudanças para um sistema de software em evolução. É essencial quando uma equipe está cooperando no desenvolvimento de software.
- A maioria dos desenvolvimentos de software é desenvolvimento *host-target*. Um IDE é usado em uma máquina *host* para desenvolver o software, que é transferido para uma máquina *target* para a execução.
- O desenvolvimento *open source* envolve a disponibilização pública do código-fonte de um sistema. Isso significa que muitas pessoas podem propor alterações e melhorias no software.

LEITURA COMPLEMENTAR

Design Patterns: Elements of Reusable Object-Oriented Software. Esse é o manual original, que apresentou os padrões de software para uma vasta comunidade. (GAMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns*: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.)

Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development, 3rd edition. Larman escreve claramente sobre o projeto orientado a objetos, assim como discute o uso da UML. Essa é uma boa introdução ao uso de padrões no processo de projeto. (LARMAN, C. *Applying UML and Patterns*: An Introduction to Object-oriented Analysis and Design and Iterative Development. 3. ed. Prentice Hall, 2004.)

Producing Open Source Software: How to Run a Successful Free Software Project. Esse livro é um guia completo para

a base do software *open source*, para as questões de licenciamento e para os aspectos práticos da execução de um projeto de desenvolvimento *open source*. (FOGEL, K. *Producing Open Source Software*: How to Run a Successful Free Software Project. O'Reilly Media Inc., 2008.)

Outras leituras sobre reúso de software são sugeridas no Capítulo 16 e no Capítulo 25 (gerenciamento de configuração).

EXERCÍCIOS

7.1 Usando a notação estruturada do Quadro 7.1, especifique casos de uso da estação meteorológica para Relatar *status* e Reconfigurar. Você deve fazer suposições razoáveis sobre a funcionalidade necessária.

7.2 Suponha que o MHC-PMS está sendo desenvolvido por meio de uma abordagem orientada a objetos. Desenhe um diagrama de caso de uso mostrando pelo menos seis possibilidades para esse sistema.

7.3 Usando a notação gráfica da UML para classes de objetos, projete as classes de objeto a seguir, identificando atributos e operações. Use sua experiência para decidir sobre os atributos e operações que devem ser associados a esses objetos.

- um telefone;
- uma impressora para um computador pessoal;
- um sistema de som estéreo pessoal;
- uma conta bancária;
- um catálogo de biblioteca.

7.4 Usando como ponto de partida os objetos da estação meteorológica identificados na Figura 7.5, identifique outros objetos que possam ser usados nesse sistema. Projete uma hierarquia de herança para os objetos que você identificou.

7.5 Desenvolva o projeto da estação meteorológica para mostrar a interação entre o subsistema de coleta de dados e os instrumentos que coletam dados meteorológicos. Use diagramas de sequência para mostrar essa interação.

7.6 Identifique possíveis objetos nos sistemas seguintes e desenvolva um projeto orientado a objetos para eles. Você pode fazer suposições razoáveis sobre os sistemas ao derivar o projeto.

- Um sistema de gerenciamento do tempo e agenda de grupo é destinado a apoiar o calendário de reuniões e compromissos de um grupo de colegas de trabalho. Quando um compromisso que envolve várias pessoas precisa ser marcado, o sistema localiza uma janela comum em cada um de suas agendas e faz o agendamento para esse momento. Se não houver janelas comuns disponíveis, ele interage com o usuário para que este possa reorganizar sua agenda pessoal a fim de criar espaço para o compromisso.
- Uma estação de abastecimento (posto de gasolina) deve ser configurada para operar de forma totalmente automatizada. Os motoristas passam seu cartão de crédito através de um leitor ligado à bomba, o cartão é verificado por comunicação com o computador da empresa de crédito e um limite de combustível é estabelecido. O motorista pode, então, colocar o combustível solicitado. Quando a liberação do combustível está completa, a mangueira da bomba é devolvida a seu coldre, e a conta do cartão de crédito do motorista é debitada no valor do combustível. O cartão de crédito é devolvido após o débito. Se o cartão for inválido, a bomba de combustível devolve-o antes de liberar o combustível.

7.7 Desenhe um diagrama de sequência que mostre as interações dos objetos em um sistema de agenda de grupo quando um grupo de pessoas está organizando uma reunião.

7.8 Desenhe um diagrama de estado da UML mostrando as possíveis mudanças de estado na agenda de grupo ou no sistema do posto de gasolina.

7.9 Usando exemplos, explique por que o gerenciamento de configuração é importante quando uma equipe está desenvolvendo um produto de software.

7.10 Uma pequena empresa desenvolveu um produto especializado que se configura especialmente para cada cliente. Novos clientes geralmente têm requisitos específicos para serem incorporados a seu sistema, e eles pagam para que estes sejam desenvolvidos. A empresa tem a oportunidade de concorrer a um novo contrato, que deverá mais que dobrar sua base de clientes. O novo cliente também deseja ter algum envolvimento com a configuração do sistema. Explique por que, nessas circunstâncias, para a empresa proprietária do software, pode ser uma boa ideia tornar o software *open source*.

REFERÊNCIAS

ABBOTT, R. Program Design by Informal English Descriptions. *Comm. ACM*, v. 26, n. 11, 983, p. 882-894.

ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. *A Pattern Language*: Towns, Building, Construction. Oxford: Oxford University Press, 1977.

BAYERSDORFER, M. Managing a Project with Open Source Components. *ACM Interactions*, v. 14, n. 6, 2007, p. 33-34.

BECK, K.; CUNNINGHAM, W. A Laboratory for Teaching Object-Oriented Thinking. Proc. OOPSLA'89 (Conference on Object-oriented Programming, Systems, Languages and Applications), *ACM Press*, 1989, p. 1-6.

BELLAGIO, D. E.; MILLIGAN, T. J. *Software Configuration Management Strategies and IBM Rational Clearcase*: A Practical Introduction. Boston: Pearson Education (IBM Press), 2005.

BUSCHMANN, F.; HENNEY, K.; SCHMIDT, D. C. *Pattern-oriented Software Architecture*. v. 4. A Pattern Language for Distributed Computing. Nova York: John Wiley & Sons, 2007a.

_______. *Pattern-oriented Software Architecture*. v. 5. On Patterns and Pattern Languages. Nova York: John Wiley & Sons, 2007b.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P. *Pattern-oriented Software Architecture*. v. 1. A System of Patterns. Nova York: John Wiley & Sons, 1996.

CARLSON, D. *Eclipse Distilled*. Boston: Addison-Wesley, 2005.

CLAYBERG, E.; RUBEL, D. *Eclipse*: Building Commercial-Quality Plug-ins. Boston: Addison Wesley, 2006.

COAD, P.; YOURDON, E. *Object-oriented Analysis*. Englewood Cliffs, NJ: Prentice Hall, 1990.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns*: Elements of Reusable Object-oriented Software. Reading, Mass.: Addison-Wesley, 1995.

HAREL, D. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Programming*, v. 8, n. 3, 1987, p. 231-274.

KIRCHER, M.; JAIN, P. *Pattern-oriented Software Architecture*. v. 3. Patterns for Resource Management. Nova York: John Wiley & Sons, 2004.

MASSOL, V. *JUnit in Action*. Greenwich, CT: Manning Publications, 2003.

PILATO, C.; COLLINS-SUSSMAN, B.; FITZPATRICK, B. *Version Control with Subversion*. Sebastopol, Calif.: O'Reilly Media Inc., 2008.

RAYMOND, E. S. *The Cathedral and the Bazaar*: Musings on Linux and Open Source by an Accidental Revolutionary. Sebastopol. Calif.: O'Reilly Media, Inc., 2001.

SCHMIDT, D.; STAL, M.; ROHNERT, H.; BUSCHMANN, F. *Pattern-oriented Software Architecture*. v. 2. Patterns for Concurrent and Networked Objects. Nova York: John Wiley & Sons, 2000.

SHLAER, S.; MELLOR, S. *Object-oriented Systems Analysis*: Modeling the World in Data. Englewood Cliffs, NJ: Yourdon Press, 1998.

St. LAURENT, A. *Understanding Open Source and Free Software Licensing*. Sebastopol, Calif.: O'Reilly Media Inc., 2004.

WIRFS-BROCK, R.; WILKERSON, B.; WEINER, L. *Designing Object-oriented Software*. Englewood Cliffs, NJ: Prentice Hall, 1990.

CAPÍTULO

1 2 3 4 5 6 7 **8** 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Testes de software

Objetivos

O objetivo deste capítulo é introduzir testes de software e processos de testes de software. Com a leitura deste capítulo, você:

- compreenderá os estágios de teste durante o desenvolvimento para os testes de aceitação por parte dos usuários de sistema;
- terá sido apresentado a técnicas que ajudam a escolher casos de teste orientados para a descoberta de defeitos de programa;
- compreenderá o desenvolvimento *test-first*, em que você projeta testes antes de escrever o código e os executa automaticamente;
- conhecerá as diferenças importantes entre teste de componentes, de sistemas e de release, e estará ciente dos processos e técnicas de teste de usuário.

Conteúdo

O teste é destinado a mostrar que um programa faz o que é proposto a fazer e para descobrir os defeitos do programa antes do uso. Quando se testa o software, o programa é executado usando dados fictícios. Os resultados do teste são verificados à procura de erros, anomalias ou informações sobre os atributos não funcionais do programa.

O processo de teste tem dois objetivos distintos:

1. Demonstrar ao desenvolvedor e ao cliente que o software atende a seus requisitos. Para softwares customizados, isso significa que deve haver pelo menos um teste para cada requisito do documento de requisitos. Para softwares genéricos, isso significa que deve haver testes para todas as características do sistema, além de suas combinações, que serão incorporadas ao release do produto.
2. Descobrir situações em que o software se comporta de maneira incorreta, indesejável ou de forma diferente das especificações. Essas são consequências de defeitos de software. O teste de defeitos preocupa-se com a eliminação de comportamentos indesejáveis do sistema, tais como panes, interações indesejáveis com outros sistemas, processamentos incorretos e corrupção de dados.

O primeiro objetivo leva a testes de validação, nos quais você espera que o sistema execute corretamente usando determinado conjunto de casos de teste que refletem o uso esperado do sistema. O segundo objetivo leva a testes de defeitos, nos quais os casos de teste são projetados para expor os defeitos. Os casos de teste na busca por defeitos podem ser

deliberadamente obscuros e não precisam refletir com precisão a maneira como o sistema costuma ser usado. Claro que não existem limites definidos entre essas duas abordagens de teste. Durante os testes de validação, você vai encontrar defeitos no sistema; durante o teste de defeitos, alguns dos testes mostrarão que o programa corresponde a seus requisitos.

O diagrama da Figura 8.1 pode ajudar a explicar as diferenças entre os testes de validação e o teste de defeitos. Pense no sistema sendo testado como uma caixa-preta. O sistema aceita entradas a partir de algum conjunto de entradas I e gera saídas em um conjunto de saídas O. Algumas das saídas estarão erradas. Estas são as saídas no conjunto O_e, geradas pelo sistema em resposta a entradas definidas no conjunto I_e. A prioridade nos testes de defeitos é encontrar essas entradas definidas no conjunto I_e, pois elas revelam problemas com o sistema. Testes de validação envolvem os testes com entradas corretas que estão fora do I_e. Estes estimulam o sistema a gerar corretamente as saídas.

Os testes não podem demonstrar se o software é livre de defeitos ou se ele se comportará conforme especificado em qualquer situação. É sempre possível que um teste que você tenha esquecido seja aquele que poderia descobrir mais problemas no sistema. Como eloquentemente afirmou Edsger Dijkstra, um dos primeiros colaboradores para o desenvolvimento da engenharia de software (DIJKSTRA et al. 1972):

Os testes podem mostrar apenas a presença de erros, e não sua ausência.

O teste é parte de um amplo processo de verificação e validação (V&V). Verificação e validação não são a mesma coisa, embora sejam frequentemente confundidas.

Barry Boehm, pioneiro da engenharia de software, expressou sucintamente a diferença entre validação e verificação (BOEHM, 1979):

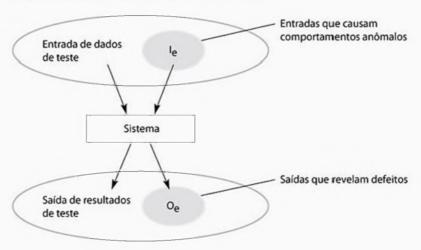
- 'Validação: estamos construindo o produto certo?'
- 'Verificação: estamos construindo o produto da maneira certa?'

Os processos de verificação e validação objetivam verificar se o software em desenvolvimento satisfaz suas especificações e oferece a funcionalidade esperada pelas pessoas que estão pagando pelo software. Esses processos de verificação iniciam-se assim que os requisitos estão disponíveis e continuam em todas as fases do processo de desenvolvimento.

O objetivo da verificação é checar se o software atende a seus requisitos funcionais e não funcionais. Validação, no entanto, é um processo mais geral. O objetivo da validação é garantir que o software atenda às expectativas do cliente. Ele vai além da simples verificação de conformidade com as especificações, pois tenta demonstrar que o software faz o que o cliente espera que ele faça. A validação é essencial porque, como já discutido no Capítulo 4, especificações de requisitos nem sempre refletem os desejos ou necessidades dos clientes e usuários do sistema.

O objetivo final dos processos de verificação e validação é estabelecer a confiança de que o software está 'pronto para seu propósito'. Isso significa que o sistema deve ser bom o suficiente para seu intuito. O nível de confiança exigido depende do propósito do sistema, das expectativas dos usuários do sistema e do atual ambiente de marketing:

1. *Finalidade do software.* O mais importante quando se fala sobre software é que ele seja confiável. Por exemplo, o nível de confiança necessário para um software ser usado para controlar um sistema crítico de segurança é muito maior do que o necessário para um protótipo que foi desenvolvido para demonstrar as ideias de novos produtos.

Figura 8.1 Um modelo de entrada-saída de teste de programa

2. *Expectativas de usuários*. Devido a suas experiências com softwares defeituosos e não confiáveis, muitos usuários têm baixas expectativas acerca da qualidade de software. Eles não se surpreendem quando o software falha. Quando um novo sistema é instalado, os usuários podem tolerar falhas, pois os benefícios do uso compensam os custos de recuperação de falhas. Nessas situações, você pode não precisar se dedicar muito a testar o software. No entanto, com o amadurecimento do software, os usuários esperam que ele se torne mais confiável e, assim, testes mais completos das próximas versões podem ser necessários.
3. *Ambiente de marketing*. Quando um sistema é comercializado, os vendedores devem levar em conta os produtos concorrentes, o preço que os clientes estão dispostos a pagar por um sistema e os prazos necessários para a entrega desse sistema. Em um ambiente competitivo, uma empresa de software pode decidir lançar um produto antes que ele tenha sido totalmente testado e depurado, pois quer ser a primeira no mercado. Se um software é muito barato, os usuários podem tolerar um baixo nível de confiabilidade.

Assim como testes de software, o processo V&V pode incluir inspeções e revisões. Eles analisam e verificam os requisitos de sistema, modelos de projeto, o código-fonte de programa e até mesmo os testes de sistema propostos. Essas são chamadas técnicas 'estáticas' de V&V, em que você não precisa executar o software para verificá-lo. A Figura 8.2 mostra as inspeções e testes de software que apoiam o V&V em diferentes estágios do processo de software. As setas indicam os estágios do processo em que as técnicas podem ser usadas.

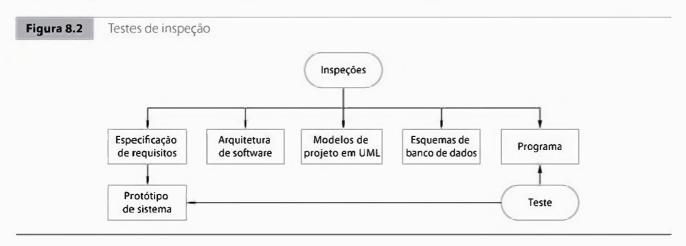
As inspeções centram-se principalmente no código-fonte de um sistema, mas qualquer representação legível do software, como seus requisitos ou modelo de projeto, pode ser inspecionada. Ao inspecionar um sistema, você usa o conhecimento do sistema, seu domínio de aplicação e a linguagem de programação ou modelagem para descobrir erros.

Existem três vantagens da inspeção de software sobre os testes:

1. Durante o teste, erros podem mascarar (esconder) outros erros. Quando um erro conduz saídas inesperadas, você nunca tem certeza se as anomalias seguintes são devidas a um novo erro ou efeitos colaterais do erro original. Como a inspeção é um processo estático, você não precisa se preocupar com as interações entre os erros. Consequentemente, uma sessão única de inspeção pode descobrir muitos erros no sistema.
2. Versões incompletas de um sistema podem ser inspecionadas sem custos adicionais. Se um programa é incompleto, você precisa desenvolver dispositivos de teste especializados para testar as partes disponíveis. Isso, obviamente, aumenta os custos de desenvolvimento do sistema.
3. Bem como a procura por defeitos de programa, uma inspeção pode considerar outros atributos de qualidade de um programa, como a conformidade com os padrões, portabilidade e manutenibilidade. Você pode procurar ineficiências, algoritmos inadequados e um estilo pobre de programação que poderiam tornar o sistema de difícil manutenção e atualização.

As inspeções de programa são uma ideia antiga, e vários estudos e experimentos demonstraram que as inspeções são mais eficazes na descoberta de defeitos do que os testes de programa. Fagan (1986) relatou que mais de 60% dos erros em um programa podem ser detectados por meio de inspeções informais de programa. No processo Cleanroom (PROWELL et al., 1999), afirma-se que mais de 90% dos defeitos podem ser descobertos em inspeções de programas.

No entanto, as inspeções não podem substituir os testes de software. As inspeções não são boas para descobrir defeitos que surgem devido a interações inesperadas entre diferentes partes de um programa, problemas de *timing* ou com o desempenho do sistema. Além disso, pode ser difícil e caro montar uma equipe de inspeção, especialmente em pequenas empresas ou grupos de desenvolvimento, já que todos os membros da equipe também podem ser desenvolvedores de

Figura 8.2 Testes de inspeção

software. No Capítulo 24 (Gerenciamento de qualidade), discuto revisões e inspeções com mais detalhes. A análise estática automatizada, em que o texto-fonte de um programa é automaticamente analisado para descobrir anomalias, é explicada no Capítulo 15. Neste capítulo, o foco está nos testes e processos de testes.

A Figura 8.3 é um modelo abstrato do processo 'tradicional' de testes, como usado no desenvolvimento dirigido a planos. Os casos de teste são especificações das entradas para o teste e da saída esperada do sistema (os resultados do teste), além de uma declaração do que está sendo testado. Os dados de teste são as entradas criadas para testar um sistema. Às vezes, os dados de teste podem ser gerados automaticamente, mas a geração automática de casos de teste é impossível, pois as pessoas que entendem o propósito do sistema devem ser envolvidas para especificar os resultados esperados. No entanto, a execução do teste pode ser automatizada. Os resultados esperados são automaticamente comparados aos resultados previstos, por isso não há necessidade de uma pessoa para procurar erros e anomalias na execução dos testes.

Geralmente, o sistema de software comercial tem de passar por três estágios de teste:

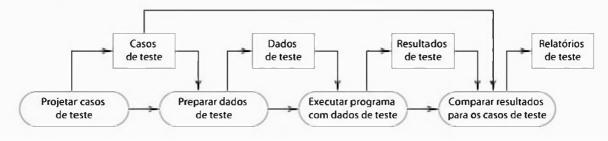
1. Testes em desenvolvimento, em que o sistema é testado durante o desenvolvimento para descobrir *bugs* e defeitos. Projetistas de sistemas e programadores podem estar envolvidos no processo de teste.
2. Testes de release, em que uma equipe de teste independente testa uma versão completa do sistema antes que ele seja liberado para os usuários. O objetivo dos testes de release é verificar se o sistema atende aos requisitos dos *stakeholders* de sistema.
3. Testes de usuário, em que os usuários ou potenciais usuários de um sistema testam o sistema em seu próprio ambiente. Para produtos de software, o 'usuário' pode ser um grupo de marketing interno, que decidirá se o software pode ser comercializado, liberado e vendido. Os testes de aceitação são um tipo de teste de usuário no qual o cliente testa formalmente o sistema para decidir se ele deve ser aceito por parte do fornecedor do sistema ou se é necessário um desenvolvimento adicional.

Na prática, o processo de teste geralmente envolve uma mistura de testes manuais e automatizados. No teste manual, um testador executa o programa com alguns dados de teste e compara os resultados com suas expectativas; ele anota e reporta as discrepâncias aos desenvolvedores do programa. Em testes automatizados, os testes são codificados em um programa que é executado cada vez que o sistema em desenvolvimento é testado. Essa forma é geralmente mais rápida que o teste manual, especialmente quando envolve testes de regressão — reexecução de testes anteriores para verificar se as alterações no programa não introduziram novos *bugs*.

O uso de testes automatizados tem aumentado consideravelmente nos últimos anos. Entretanto, os testes nunca poderão ser totalmente automatizados, já que testes automáticos só podem verificar se um programa faz aquilo a que é proposto. É praticamente impossível usar testes automatizados para testar os sistemas que dependem de como as coisas estão (por exemplo, uma interface gráfica de usuário), ou para testar se um programa não tem efeitos colaterais indesejados.

8.1 Testes de desenvolvimento

Testes de desenvolvimento incluem todas as atividades de testes que são realizadas pela equipe de desenvolvimento do sistema. O testador do software geralmente é o programador que o desenvolveu, embora nem sempre seja assim. Alguns processos de desenvolvimento usam programadores/testadores em pares (CUSAMANO e SELBY, 1998), nos quais cada programador tem um testador associado para desenvolver os testes e ajudar no processo. Para sistemas críticos, um processo mais formal pode ser usado por um grupo de testes independente

Figura 8.3 Um modelo do processo de teste de software

dentro da equipe de desenvolvimento. Eles são responsáveis pelo desenvolvimento de testes e pela manutenção de registros detalhados de seus resultados.

Durante o desenvolvimento, o teste pode ocorrer em três níveis de granularidade:

1. Teste unitário, em que as unidades individuais de programa ou classes de objetos são testadas individualmente. Testes unitários devem centrar-se em testar a funcionalidade dos objetos ou métodos.
2. Teste de componentes, em que várias unidades individuais são integradas para criar componentes compostos. Testes de componentes devem centrar-se em testar as interfaces dos componentes.
3. Teste de sistema, em que alguns ou todos os componentes de um sistema estão integrados e o sistema é testado como um todo. O teste de sistema deve centrar-se em testar as interações entre os componentes.

Testes de desenvolvimento são essencialmente um processo de teste de defeitos, em que o objetivo do teste é descobrir *bugs* no software. Normalmente, são intercalados com a depuração — o processo de localizar problemas com o código e alterar o programa para corrigir esses problemas.

8.1.1 Teste unitário

O teste unitário é o processo de testar os componentes de programa, como métodos ou classes de objeto. As funções individuais ou métodos são o tipo mais simples de componente. Seus testes devem ser chamadas para essas rotinas com parâmetros diferentes de entrada. Você pode usar as abordagens para projeto de casos de teste (discutidas na Seção 8.1.2), para projetar testes de funções ou métodos.

Quando você está testando as classes de objeto, deve projetar os testes para fornecer uma cobertura de todas as características do objeto. Isso significa que você deve:

- Testar todas as operações associadas ao objeto;
- Definir e verificar o valor de todos os atributos associados ao objeto;
- Colocar o objeto em todos os estados possíveis, o que significa simular todos os eventos que causam mudanças de estado.

Considere, por exemplo, o objeto EstaçãoMeteorológica do exemplo discutido no Capítulo 7. A interface desse objeto é mostrada na Figura 8.4. Ela tem um único atributo, que é seu identificador. Este é uma constante, definida quando a estação meteorológica é instalada. Portanto, você só precisa de um teste que verifique se ele foi configurado corretamente. Você precisa definir casos de teste para todos os métodos associados ao objeto, como relatarClima, relatarStatus etc. Preferencialmente, você deve testar métodos de forma isolada, mas, em alguns casos, algumas sequências de teste são necessárias. Por exemplo, para testar o método que desliga os instrumentos da estação meteorológica (desligar), você precisa ter executado o método de reiniciar.

A generalização ou herança faz testes de classes de objeto mais complicados. Você não pode simplesmente testar uma operação na classe em que ela está definida e assumir que funcionará corretamente nas subclasses que herdam a operação. A operação que é herdada pode fazer suposições sobre outras operações e atributos. Essas operações podem não ser válidas em algumas subclasses que herdam a operação. Portanto, é necessário testar a operação herdada em todos os contextos de uso.

Para testar os estados da estação meteorológica, usamos um modelo de estado, como o mostrado no capítulo anterior, na Figura 7.7. Usando esse modelo, é possível identificar as sequências de transições de estado que pre-

Figura 8.4 A interface do objeto EstaçãoMeteorológica

EstaçãoMeteorológica
identificador
relatarClima ( ) relatarStatus ( ) economizarEnergia (instrumentos) controlarRemoto (comandos) reconfigurar (comandos) reiniciar (instrumentos) desligar (instrumentos)

cisam ser testadas e definir as sequências de eventos para forçar essas transições. Em princípio, você deve testar cada sequência de transição de estado possível, embora na prática isso possa ser muito custoso. Exemplos de sequências de estado que devem ser testados na estação meteorológica incluem:

Desligar → Executar → Desligar

Configurar → Executar → Testar → Transmitir → Executar

Executar → Coletar → Executar → Resumir → Transmitir → Executar

Sempre que possível, você deve automatizar os testes unitários. Em testes unitários automatizados, pode-se usar um *framework* de automação de teste (como JUnit) para escrever e executar testes de seu programa. *Frameworks* de testes unitários fornecem classes de teste genéricas que você pode estender para criar casos de teste específicos. Eles podem, então, executar todos os testes que você implementou e informar, muitas vezes por meio de alguma interface gráfica, sobre o sucesso ou o fracasso dos testes. Um conjunto inteiro de testes frequentemente pode ser executado em poucos segundos; assim, é possível executar todos os testes cada vez que é feita uma alteração no programa.

Um teste automatizado tem três partes:

1. Uma parte de configuração, em que você inicia o sistema com o caso de teste, ou seja, as entradas e saídas esperadas.
2. Uma parte de chamada, quando você chama o objeto ou método a ser testado.
3. Uma parte de afirmação, em que você compara o resultado da chamada com o resultado esperado. Se a afirmação avaliada for verdadeira, o teste foi bem-sucedido; se for falsa, ele falhou.

Às vezes, o objeto que você está testando tem dependências em outros objetos que podem não ter sido escritos ou que atrasam o processo de teste quando são usados. Por exemplo, se o objeto chama um banco de dados, isso pode implicar um processo lento de instalação antes que ele possa ser usado. Nesses casos, você pode decidir usar um *mock object*. *Mock objects* são objetos com a mesma interface que os objetos externos usados para simular sua funcionalidade. Portanto, um *mock object* que simula um banco de dados pode ter apenas uns poucos itens organizados em um vetor. Eles podem ser acessados rapidamente, sem os *overheads* de chamar um banco de dados e acessar os discos. Da mesma forma, *mock objects* podem ser usados para simular operações anormais ou eventos raros. Por exemplo, se o sistema se destina a agir em determinados momentos do dia, seu *mock object* pode simplesmente retornar naqueles momentos, independentemente do horário real.

8.1.2 Escolha de casos de teste unitário

O teste é custoso e demorado, por isso é importante que você escolha casos efetivos de teste unitário. A efetividade, nesse caso, significa duas coisas:

1. Os casos de teste devem mostrar que, quando usado como esperado, o componente que você está testando faz o que ele é proposto a fazer.
2. Se houver defeitos nos componentes, estes devem ser revelados por casos de teste.

Você deve, portanto, escrever dois tipos de casos de teste. O primeiro deve refletir o funcionamento normal de um programa e deve mostrar que o componente funciona. Por exemplo, se você está testando um componente que cria e inicia um novo registro de paciente, seu caso de teste deve mostrar que o registro existe no banco de dados e que seus campos foram criados como especificados. O outro tipo de caso de teste deve ser baseado em testes de experiência, dos quais surgem os problemas mais comuns. Devem-se usar entradas anormais para verificar que estes são devidamente processados e que não fazem o componente falhar.

Discuto, aqui, duas estratégias que podem ser eficazes para ajudar você a escolher casos de teste. São elas:

1. Teste de partição, em que você identifica os grupos de entradas que possuem características comuns e devem ser tratados da mesma maneira. Você deve escolher os testes dentro de cada um desses grupos.
2. Testes baseados em diretrizes, em que você usa as diretrizes de testes para escolher casos de teste. Essas diretrizes refletem a experiência anterior dos tipos de erros que os programadores cometem frequentemente no desenvolvimento de componentes.

É comum os dados de entrada e os resultados de saída de um software caírem em diferentes classes com características comuns. Exemplos dessas classes são números positivos, números negativos e seleções no *menu*. Os programas geralmente se comportam de forma comparável para todos os membros de uma classe. Ou seja, se

você testar um programa que faz um cálculo e requer dois números positivos, você deve esperar que o programa se comporte da mesma forma para todos os números positivos.

Devido a esse comportamento equivalente, essas classes são também chamadas de partições ou domínios de equivalência (BEZIER, 1990). Uma abordagem sistemática para projetar casos de teste baseia-se na identificação de todas as partições de entrada e saída para um sistema ou componente. Os casos de teste são projetados para que as entradas ou saídas estejam dentro dessas partições. Um teste de partição pode ser usado para projetar casos de teste para ambos, sistemas e componentes.

Na Figura 8.5, a elipse maior sombreada, à esquerda, representa o conjunto de todas as entradas possíveis para o programa que está sendo testado. As elipses menores (não sombreadas) representam partições de equivalência. Um programa que está sendo testado deve processar todos os membros de uma partição de equivalência de entrada da mesma forma. As partições de equivalência de saída são partições dentro das quais todas as saídas têm algo em comum. Às vezes, não há um mapeamento 1:1 entre as partições de equivalência de entrada e de saída. No entanto, isso nem sempre é o caso. Você pode precisar definir uma partição de equivalência de entrada separada, na qual a única característica comum das entradas é que geram saídas dentro da mesma partição de saída. A área sombreada na elipse esquerda representa as entradas inválidas. A área sombreada na elipse da direita representa as exceções que podem ocorrer (ou seja, respostas às entradas inválidas).

Depois de ter identificado um conjunto de partições, você escolhe os casos de teste de cada uma. Uma boa prática para a seleção do caso de teste é escolher casos de teste sobre os limites das partições, além de casos perto do ponto médio da partição. A razão disso é que, no desenvolvimento de um sistema, os projetistas e programadores tendem a considerar os valores típicos de entrada. Você pode testá-los escolhendo o ponto médio da partição. Os valores-limite são frequentemente atípicos (por exemplo, zero pode comportar-se de maneira diferente de outros números não negativos) e por vezes são negligenciados pelos desenvolvedores. Falhas de programa ocorrem frequentemente ao se processarem esses valores atípicos.

As partições são identificadas usando-se a especificação de programa ou a documentação de usuário, bem como a partir da experiência com a qual você prevê as classes de valor de entrada prováveis de detectar erros. Por exemplo, digamos que uma especificação de programa defina que o programa aceita entradas de 4 a 8 que sejam valores inteiros de cinco dígitos superiores a 10.000. Você pode usar essas informações para identificar as partições de entrada e de possíveis valores de entrada de teste. Estes são mostrados na Figura 8.6.

Quando você usa a especificação de um sistema para identificar as partições de equivalência, isso é chamado 'teste de caixa-preta'. Nesse caso, você não precisa de nenhum conhecimento de como funciona o sistema. No entanto, pode ser útil para suplementar os testes de caixa-preta um 'teste de caixa-branca', em que você pode olhar o código do programa para encontrar outros testes possíveis. Por exemplo, seu código pode incluir exceções para lidar com entradas incorretas. Você pode usar esse conhecimento para identificar as 'partições de exceção' — diferentes intervalos, nos quais o mesmo tratamento de exceção deve ser aplicado.

Particionamento de equivalência é uma abordagem eficaz para testes, pois ajuda a explicar os erros que os programadores costumam cometer ao processar as entradas nos limites das partições. Você também pode usar

Figura 8.5 Particionamento de equivalência

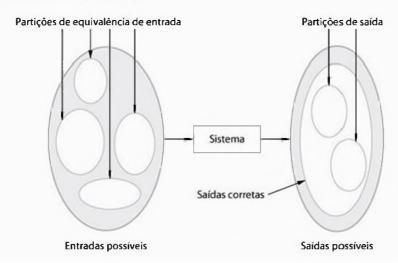
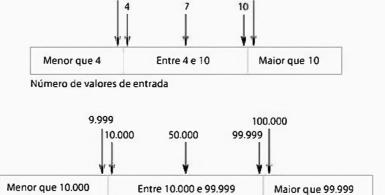



Figura 8.6 Partições de equivalência

diretrizes de teste para ajudar a escolher casos de teste. Diretrizes encapsulam o conhecimento de quais tipos de casos de teste são eficazes para descobrir erros. Por exemplo, quando você está testando programas com sequências, vetores ou listas, as diretrizes que podem ajudar a revelar defeitos incluem:

1. Testar software com as sequências que possuem apenas um valor. Programadores, em geral, pensam em sequências compostas de vários valores e, às vezes, incorporam essa hipótese em seus programas. Consequentemente, se apresentado com uma sequência de valor único, um programa pode não funcionar corretamente.
2. Usar sequências diferentes de tamanhos diferentes em testes diferentes. Isso diminui as chances de um programa com defeitos produzir acidentalmente uma saída correta, por causa de alguma característica acidental na entrada.
3. Derivar testes de modo que o primeiro elemento, o do meio e o último da sequência sejam acessados. Essa abordagem revela problemas nos limites de partição.

O livro de Whittaker (2002) inclui muitos exemplos de diretrizes que podem ser usadas no projeto de casos de teste. Algumas das diretrizes mais gerais que ele sugere são:

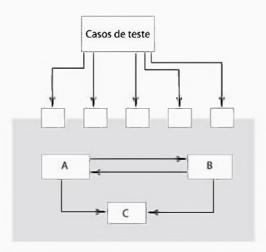
- Escolha entradas que forcem o sistema a gerar todas as mensagens de erro;
- Projete entradas que causem *overflow* de *buffers* de entrada;
- Repita a mesma entrada ou uma série de entradas inúmeras vezes;
- Obrigue a geração de saídas inválidas;
- Obrigue os resultados de cálculos a serem muito grandes ou muito pequenos.

Conforme ganha experiência com o teste, você pode desenvolver suas próprias diretrizes de como escolher casos mais eficazes. Na próxima seção deste capítulo, dou mais exemplos de diretrizes de testes.

8.1.3 Teste de componente

Frequentemente, os componentes do software são compostos de diversos objetos que interagem. Por exemplo, no sistema de estação meteorológica, o componente de reconfiguração inclui objetos que lidam com cada aspecto da reconfiguração. Você pode acessar a funcionalidade desses objetos por meio da interface de componente definida. Testes de componentes compostos devem centrar-se em mostrar que a interface de componente se comporta de acordo com sua especificação. Você pode assumir que os testes unitários sobre os objetos individuais dentro do componente já foram concluídos.

A Figura 8.7 ilustra a ideia de teste de interface de componente. Suponha que os componentes A, B e C foram integrados para criar um componente maior ou subsistema. Os casos de teste não são aplicados aos componentes individuais, mas sim à interface de componente criada pela combinação desses componentes. Erros de interface no componente composto podem não ser detectáveis por meio de testes em objetos individuais, pois esses erros resultam de interações entre os objetos do componente.

Figura 8.7 Teste de interface

Existem diferentes tipos de interface entre os componentes de programa e, consequentemente, diferentes tipos de erros de interface que podem ocorrer:

1. *Interfaces de parâmetro*. São as interfaces nas quais as referências de dados ou, às vezes, de função, são passadas de um componente para outro. Métodos de um objeto têm uma interface de parâmetro.
2. *Interfaces de memória compartilhada*. São as interfaces nas quais um bloco de memória é compartilhado entre os componentes. Os dados são colocados na memória por um subsistema e recuperados a partir daí por outros subsistemas. Esse tipo de interface é frequentemente usado em sistemas embutidos, em que os sensores criam dados que são recuperados e processados por outros componentes do sistema.
3. *Interfaces de procedimento*. São as interfaces nas quais um componente encapsula um conjunto de procedimentos que podem ser chamados por outros componentes. Objetos e componentes reusáveis têm esse tipo de interface.
4. *Interface de passagem de mensagem*. São as interfaces nas quais um componente solicita um serviço de outro componente, passando-lhe uma mensagem. Uma mensagem de retorno inclui os resultados da execução do serviço. Alguns sistemas orientados a objetos têm esse tipo de interface, como nos sistemas cliente-servidor.

Erros de interface são uma das formas mais comuns de erros em sistemas complexos (LUTZ, 1993). Esses erros são classificados em três classes:

- *Mau uso de interface*. Um componente chamador chama outro componente e comete um erro no uso de sua interface. Esse tipo de erro é comum com interfaces de parâmetro, em que os parâmetros podem ser de tipo errado ou ser passados na ordem errada, ou o número errado de parâmetros pode ser passado.
- *Mau-entendimento de interface*. Um componente chamador desconhece a especificação da interface do componente chamado e faz suposições sobre seu comportamento. O componente chamado não se comporta conforme o esperado, causando um comportamento inesperado no componente de chamada. Por exemplo, um método de busca binária pode ser chamado com um parâmetro que é um vetor não ordenado. A busca então falharia.
- *Erros de timing*. Eles ocorrem em sistemas em tempo real que usam uma memória compartilhada ou uma interface de passagem de mensagens. O produtor e o consumidor de dados podem operar em velocidades diferentes. A menos que se tome um cuidado especial no projeto da interface, o consumidor pode acessar uma informação desatualizada, porque o produtor da informação não atualizou as informações da interface compartilhada.

Testes para defeitos de interface são difíceis, pois alguns defeitos de interface só podem manifestar-se sob condições não usuais. Por exemplo, digamos que um objeto implementa uma fila como uma estrutura de dados de comprimento fixo. Um objeto chamador pode supor que a fila é implementada como uma estrutura de dados infinita e pode não verificar o *overflow* de fila quando um item for inserido. Essa condição só pode ser detectada durante os testes, projetando casos de teste que forçam a fila a transbordar e causar estouro, o que corrompe o comportamento do objeto de forma detectável.

Outro problema pode surgir por causa das interações entre defeitos em diferentes módulos ou objetos. Defeitos em um objeto só podem ser detectados quando outro objeto se comporta de maneira inesperada. Por exemplo, um objeto pode chamar outro para receber algum serviço e assumir que a resposta está correta. Se o serviço chamado está defeituoso de algum modo, o valor retornado pode ser válido, porém incorreto. Isso não é imediatamente detectado, mas só se torna evidente quando algum processamento posterior dá errado.

Algumas diretrizes gerais para os testes de interface são:

1. Examine o código a ser testado e liste, explicitamente, cada chamada para um componente externo. Projete um conjunto de testes em que os valores dos parâmetros para os componentes externos estão nos extremos de suas escalas. Esses valores extremos são mais suscetíveis a revelar inconsistências de interface.
2. Nos casos em que os ponteiros são passados por meio de uma interface, sempre teste a interface com os parâmetros de ponteiros nulos.
3. Nos casos em que um componente é chamado por meio de uma interface de procedimento, projete testes que deliberadamente causem uma falha no componente. Diferentes hipóteses de falhas são um dos equívocos mais comuns de especificação.
4. Use testes de estresse em sistemas de passagem de mensagem. Isso significa que você deve projetar testes que gerem muito mais mensagens do que seria provável ocorrer na prática. Essa é uma maneira eficaz de revelar problemas de *timing*.
5. Nos casos em que vários componentes interagem por meio de memória compartilhada, desenvolva testes que variam a ordem em que esses componentes são ativados. Esses testes podem revelar as suposições implícitas feitas pelo programador sobre a ordem na qual os dados compartilhados são produzidos e consumidos.

Inspeções e avaliações podem ser mais eficazes do que os testes para descobrir erros de interface. As inspeções podem concentrar-se em interfaces de componentes e questões sobre o comportamento da interface levantadas durante o processo de inspeção. Uma linguagem fortemente tipada, como Java, permite que o compilador apreenda muitos erros de interface. Analisadores estáticos (veja o Capítulo 15) podem detectar uma vasta gama de erros de interface.

8.1.4 Teste de sistema

O teste de sistema, durante o desenvolvimento, envolve a integração de componentes para criação de uma versão do sistema e, em seguida, o teste do sistema integrado. O teste de sistema verifica se os componentes são compatíveis, se interagem corretamente e transferem os dados certos no momento certo, por suas interfaces. Certamente, sobrepõem-se ao teste de componente, mas existem duas diferenças importantes:

1. Durante o teste de sistema, os componentes reusáveis que tenham sido desenvolvidos separadamente e os sistemas de prateleira podem ser integrados com componentes recém-desenvolvidos. Então, o sistema completo é testado.
2. Nesse estágio, componentes desenvolvidos por diferentes membros da equipe ou grupos podem ser integrados. O teste de sistema é um processo coletivo, não individual. Em algumas empresas, o teste de sistema pode envolver uma equipe independente, sem participação de projetistas e programadores.

Quando você integra componentes para criar um sistema, obtém um comportamento emergente. Isso significa que alguns elementos da funcionalidade do sistema só se tornam evidentes quando colocados juntos. Esse comportamento emergente pode ser planejado e precisa ser testado. Por exemplo, você pode integrar um componente de autenticação com um componente que atualiza as informações. Assim, você tem uma característica do sistema que restringe as informações de atualização para usuários autorizados. Contudo, às vezes, o comportamento emergente não é planejado ou desejado. É preciso desenvolver testes que verifiquem se o sistema está fazendo apenas o que ele supostamente deve fazer.

Portanto, o teste do sistema deve centrar-se em testar as interações entre os componentes e objetos que compõem um sistema. Você também pode testar componentes ou sistemas reusáveis para verificar se, quando integrados com novos componentes, eles funcionam como o esperado. Esse teste de interação deve descobrir *bugs* de componente que só são revelados quando um componente é usado por outros componentes do sistema. O teste de interação também ajuda a encontrar equívocos dos desenvolvedores de componentes sobre outros componentes do sistema.

Por causa de seu foco na interação, o teste baseado em caso de uso é uma abordagem eficaz para testes de sistema. Normalmente, cada caso de uso é implementado por vários componentes ou objetos do sistema. Testar os casos de uso força essas interações a ocorrerem. Se você desenvolveu um diagrama de sequência para modelar a implementação dos casos de uso, poderá ver os objetos ou componentes envolvidos na interação.

Para ilustrar isso, uso um exemplo do sistema da estação meteorológica no deserto, em que é solicitado à estação meteorológica o relatório resumido de dados meteorológicos para um computador remoto. Esse caso de uso está descrito no Quadro 7.1. A Figura 8.8 (que é uma cópia da Figura 7.6) mostra a sequência de operações na estação meteorológica quando esta responde a um pedido para coletar dados para o sistema de mapeamento. Você pode usar esse diagrama para identificar as operações que serão testadas e para ajudar a projetar os casos de teste para a execução. Portanto, a emissão de um pedido de um relatório resultará na execução do seguinte *thread* de métodos:

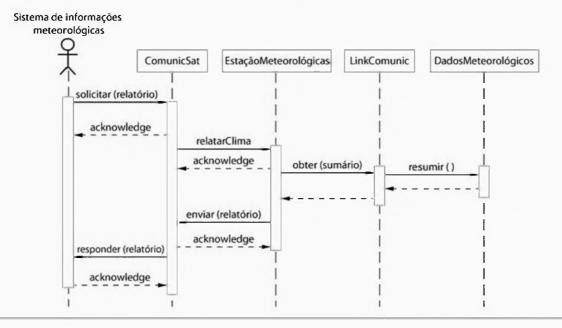
ComunicSat:solicitar → EstaçãoMeteorológica:relatarClima → LinkComunic:Obter(sumário) → DadosMeteorológicos:resumir

O diagrama de sequência ajuda a projetar os casos de teste específicos que você necessita, pois mostra quais são as entradas necessárias e que saídas são criadas:

1. Uma entrada de um pedido de relatório deve ter um *acknowledge* associado. Um relatório deve ser devolvido a partir da solicitação. Durante o teste, você deve criar dados resumidos que possam ser usados para verificar se o relatório está organizado corretamente.
2. Uma solicitação de entrada para um relatório na EstaçãoMeteorológica resulta em um relatório resumido que será gerado. Você pode fazer esse teste de forma isolada, criando dados brutos correspondentes ao resumo que você preparou para o teste de ComunicSat e verificar se o objeto EstaçãoMeteorológica produz essa síntese corretamente. Esses dados brutos também são usados para testar o objeto DadosMeteorológicos.

É claro que na Figura 8.8 eu simplifiquei o diagrama de sequência, para que ele não mostre as exceções. Um teste completo de caso de uso/cenário deve levar em consideração as exceções e garantir que os objetos as tratem corretamente.

Para a maioria dos sistemas, é difícil saber o quanto o teste de sistema é essencial e quando você deve parar de testar. É impossível fazer testes exaustivos, em que cada sequência possível de execução do programa seja testada. Assim, os testes precisam ser baseados em um subconjunto possível de casos de teste. Idealmente, as empresas de software deveriam ter políticas para a escolha desse grupo. Essas políticas podem ser baseadas em políticas de teste gerais, como uma política em que todas as declarações do programa devem ser executadas pelo menos uma vez. Como alternativa, podem basear-se na experiência de uso do sistema e centrar-se em testes de características do sistema operacional. Por exemplo:

Figura 8.8 Diagrama de sequência de coletar dados meteorológicos

1. Todas as funções do sistema acessadas por meio de *menus* devem ser testadas.
2. Combinações de funções (por exemplo, a formatação do texto) acessadas por meio de *menu* devem ser testadas.
3. Nos casos em que a entrada do usuário é fornecida, todas as funções devem ser testadas com entradas corretas e incorretas.

A partir de experiências com importantes produtos de software, como processadores de texto ou planilhas, fica evidente que durante o teste de produtos costumam ser usadas diretrizes semelhantes. Quando os recursos do software são usados de forma isolada, eles trabalham normalmente. Como explica Whittaker (2002), os problemas surgem quando as combinações das características menos usadas não são testadas em conjunto. Ele dá o exemplo de como, em um processador de texto comumente usado, usar as notas de rodapé com um *layout* de várias colunas causa erros no *layout* do texto.

Testes automatizados do sistema geralmente são mais difíceis do que testes automatizados de unidades ou de componentes. Testes automatizados de unidade baseiam-se em prever as saídas, e, em seguida, codificar essas previsões em um programa. A previsão é, então, comparada com o resultado. No entanto, o mais importante na aplicação de um sistema pode ser a geração de saídas que sejam grandes ou que não possam ser facilmente previstas. Você pode ser capaz de analisar uma saída e verificar sua credibilidade sem necessariamente ser capaz de criá-la com antecipação.

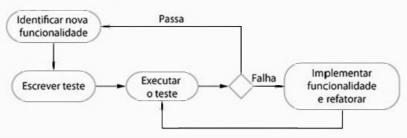
8.2 Desenvolvimento dirigido a testes

O desenvolvimento dirigido a testes (TDD, do inglês *Test-Driven Development*) é uma abordagem para o desenvolvimento de programas em que se intercalam testes e desenvolvimento de código (BECK, 2002; JEFFRIES e MELNIK, 2007). Essencialmente, você desenvolve um código de forma incremental, em conjunto com um teste para esse incremento. Você não caminha para o próximo incremento até que o código desenvolvido passe no teste. O desenvolvimento dirigido a testes foi apresentado como parte dos métodos ágeis, como o Extreme Programming. No entanto, ele também pode ser usado em processos de desenvolvimento dirigido a planos.

O processo fundamental de TDD é mostrado na Figura 8.9. As etapas do processo são:

1. Você começa identificando o incremento de funcionalidade necessário. Este, normalmente, deve ser pequeno e implementável em poucas linhas de código.
2. Você escreve um teste para essa funcionalidade e o implementa como um teste automatizado. Isso significa que o teste pode ser executado e relatará se passou ou falhou.
3. Você, então, executa o teste, junto com todos os outros testes implementados. Inicialmente, você não terá implementado a funcionalidade, logo, o novo teste falhará. Isso é proposital, pois mostra que o teste acrescenta algo ao conjunto de testes.
4. Você, então, implementa a funcionalidade e executa novamente o teste. Isso pode envolver a refatoração do código existente para melhorá-lo e adicionar um novo código sobre o que já está lá.
5. Depois que todos os testes forem executados com sucesso, você caminha para implementar a próxima parte da funcionalidade.

Um ambiente de testes automatizados, como o ambiente JUnit, que suporta o teste de programa Java (MASSOL e HUSTED, 2003), é essencial para o TDD. Como o código é desenvolvido em incrementos muito pequenos, você precisa ser capaz de executar todos os testes cada vez que adicionar funcionalidade ou refatorar o programa.

Figura 8.9 Desenvolvimento dirigido a testes

Portanto, os testes são embutidos em um programa separado que os executa e invoca o sistema que está sendo testado. Usando essa abordagem, é possível rodar centenas de testes separados em poucos segundos.

Um argumento forte a favor do desenvolvimento dirigido a testes é que ele ajuda os programadores a clarear suas ideias sobre o que um segmento de código supostamente deve fazer. Para escrever um teste, você precisa entender a que ele se destina, e como esse entendimento faz que seja mais fácil escrever o código necessário. Certamente, se você tem conhecimento ou compreensão incompleta, o desenvolvimento dirigido a testes não ajudará. Se você não sabe o suficiente para escrever os testes, não vai desenvolver o código necessário. Por exemplo, se seu cálculo envolve divisão, você deve verificar se não está dividindo o número por zero. Se você se esquecer de escrever um teste para isso, então o código para essa verificação nunca será incluído no programa.

Além de um melhor entendimento do problema, outros benefícios do desenvolvimento dirigido a testes são:

1. *Cobertura de código*. Em princípio, todo segmento de código que você escreve deve ter pelo menos um teste associado. Portanto, você pode ter certeza de que todo o código no sistema foi realmente executado. Cada código é testado enquanto está sendo escrito; assim, os defeitos são descobertos no início do processo de desenvolvimento.
2. *Teste de regressão*. Um conjunto de testes é desenvolvido de forma incremental enquanto um programa é desenvolvido. Você sempre pode executar testes de regressão para verificar se as mudanças no programa não introduziram novos *bugs*.
3. *Depuração simplificada*. Quando um teste falha, a localização do problema deve ser óbvia. O código recém-escrito precisa ser verificado e modificado. Você não precisa usar as ferramentas de depuração para localizar o problema. Alguns relatos de uso de desenvolvimento dirigido a testes sugerem que, em desenvolvimento dirigido a testes, quase nunca é necessário usar um sistema automatizado de depuração (MARTIN, 2007).
4. *Documentação de sistema*. Os testes em si mesmos agem como uma forma de documentação que descreve o que o código deve estar fazendo. Ler os testes pode tornar mais fácil a compreensão do código.

Um dos benefícios mais importantes de desenvolvimento dirigido a testes é que ele reduz os custos dos testes de regressão. O teste de regressão envolve a execução de conjuntos de testes que tenham sido executados com sucesso, após as alterações serem feitas em um sistema. O teste de regressão verifica se essas mudanças não introduziram novos *bugs* no sistema e se o novo código interage com o código existente conforme o esperado. O teste de regressão é muito caro e geralmente impraticável quando um sistema é testado manualmente, pois os custos com tempo e esforço são muito altos. Em tais situações, você precisa tentar escolher os testes mais relevantes para executar novamente, e é fácil perder testes importantes.

No entanto, testes automatizados, fundamentais para o desenvolvimento *test-first*, reduzem drasticamente os custos com testes de regressão. Os testes existentes podem ser executados novamente de forma rápida e barata. Após se fazer uma mudança para um sistema em desenvolvimento *test-first*, todos os testes existentes devem ser executados com êxito antes de qualquer funcionalidade ser adicionada. Como um programador, você precisa ter certeza de que a nova funcionalidade não tenha causado ou revelado problemas com o código existente.

O desenvolvimento dirigido a testes é de maior utilidade no desenvolvimento de softwares novos, em que a funcionalidade seja implementada no novo código ou usando bibliotecas-padrão já testadas. Se você estiver reusando componentes de código ou sistemas legados grandes, você precisa escrever testes para esses sistemas como um todo. O desenvolvimento dirigido a testes também pode ser ineficaz em sistemas multi-*threaded*. Os *threads* diferentes podem ser intercalados em tempos diferentes, em execuções diferentes, e isso pode produzir resultados diferentes.

Se você usa o desenvolvimento dirigido a testes, ainda precisa de um processo de teste de sistema para validar o sistema, isto é, para verificar se ele atende aos requisitos de todos os *stakeholders*. O teste de sistema também testa o desempenho, a confiabilidade, e verifica se o sistema não faz coisas que não deveria, como produzir resultados indesejados etc. Andrea (2007) sugere como ferramentas de teste podem ser estendidas para integrar alguns aspectos do teste de sistema com TDD.

O desenvolvimento dirigido a testes revelou-se uma abordagem de sucesso para projetos de pequenas e médias empresas. Geralmente, os programadores que adotaram essa abordagem estão satisfeitos com ela e acham que é uma maneira mais produtiva de desenvolver softwares (JEFFRIES e MELNIK, 2007). Em alguns experimentos, foi mostrado que essa abordagem gera melhorias na qualidade do código; em outros, os resultados foram inconclusivos; no entanto, não existem evidências de que o TDD leve à redução da qualidade de código.

8.3 Testes de release

Teste de release é o processo de testar um release particular de um sistema que se destina para uso fora da equipe de desenvolvimento. Geralmente, o release de sistema é para uso dos clientes e usuários. Em um projeto complexo, no entanto, ele pode ser para as outras equipes que estão desenvolvendo sistemas relacionados. Para produtos de software, o release pode ser para gerenciamento de produtos que, em seguida, o prepara para a venda.

Existem duas diferenças importantes entre o teste de release e o teste de sistema durante o processo de desenvolvimento:

1. Uma equipe separada, que não esteve envolvida no desenvolvimento do sistema, deve ser responsável pelo teste de release.
2. Testes de sistema feitos pela equipe de desenvolvimento devem centrar-se na descoberta de *bugs* no sistema (teste de defeitos). O objetivo do teste de release é verificar se o sistema atende a seus requisitos e é bom o suficiente para uso externo (teste de validação).

O objetivo principal do processo de teste de release é convencer o fornecedor do sistema de que esse sistema é bom o suficiente para uso. Se assim for, o sistema poderá ser lançado como um produto ou entregue aos clientes. Portanto, o teste de release precisa mostrar que o sistema oferece a funcionalidade, o desempenho e a confiança especificados e que não falhará durante o uso normal. Deve levar em conta todos os requisitos de sistema, e não apenas os requisitos de usuários finais do sistema.

Teste de release costuma ser um processo de teste de caixa-preta, no qual os testes são derivados da especificação de sistema. O sistema é tratado como uma caixa-preta cujo comportamento só pode ser determinado por meio do estudo das entradas e saídas relacionadas. Outro nome para isso é 'teste funcional', assim chamado porque o testador só está preocupado com a funcionalidade, e não com a implementação do software.

8.3.1 Testes baseados em requisitos

Um dos princípios gerais das boas práticas de engenharia de requisitos é que os requisitos devem ser testáveis, isto é, o requisito deve ser escrito de modo que um teste possa ser projetado para ele. Um testador pode, então, verificar se o requisito foi satisfeito. Testes baseados em requisitos, portanto, são uma abordagem sistemática para projeto de casos de teste em que você considera cada requisito e deriva um conjunto de testes para eles. Testes baseados em requisitos são mais uma validação do que um teste de defeitos — você está tentando demonstrar que o sistema implementou adequadamente seus requisitos.

Por exemplo, considere os requisitos relacionados ao MHC-PMS (apresentado no Capítulo 1), preocupados com a verificação de alergias a medicamentos:

> *Se é sabido que um paciente é alérgico a algum medicamento específico, uma prescrição para esse medicamento deve resultar em uma mensagem de aviso a ser emitida ao usuário do sistema.*
>
> *Se um médico opta por ignorar um aviso de alergia, ele deve fornecer uma razão pela qual esse aviso foi ignorado.*

Para verificar se esses requisitos foram satisfeitos, pode ser necessário desenvolver vários testes relacionados:

1. Defina um registro de paciente sem alergias conhecidas. Prescreva medicação para alergias que são conhecidas. Verifique se uma mensagem de alerta não é emitida pelo sistema.
2. Defina um registro de paciente com alergia. Prescreva a medicação à qual o paciente é alérgico e verifique se o aviso é emitido pelo sistema.
3. Defina um registro de paciente no qual se registram alergias a dois ou mais medicamentos. Prescreva ambos os medicamentos separadamente e verifique se o aviso correto para cada um é emitido.
4. Prescreva dois medicamentos a que o paciente é alérgico. Verifique se os dois avisos são emitidos corretamente.
5. Prescreva um medicamento que emita um aviso e ignore esse aviso. Verifique se o sistema exige que o usuário forneça informações que expliquem por que o aviso foi ignorado.

A partir disso, você pode ver que testar um requisito não significa simplesmente escrever um único teste. Normalmente, você precisa escrever vários testes para garantir a cobertura dos requisitos. Você também deve manter registros de rastreabilidade de seus testes baseados em requisitos, que ligam os testes aos requisitos específicos que estão sendo testados.

8.3.2 Testes de cenário

Teste de cenário é uma abordagem de teste de release em que você imagina cenários típicos de uso e os usa para desenvolver casos de teste para o sistema. Um cenário é uma história que descreve uma maneira de usar o sistema. Cenários devem ser realistas, e usuários reais do sistema devem ser capazes de se relacionar com eles. Se você já usou cenários como parte do processo de engenharia de requisitos (descritos no Capítulo 4), então você é capaz de reusá-los ao testar cenários.

Em um breve artigo sobre testes de cenário, Kaner (2003) sugere que um teste de cenário deve ser uma história narrativa crível e bastante complexa. Deve motivar os *stakeholders*, ou seja, eles devem se relacionar com o cenário e acreditar que é importante que o sistema passe no teste. Kaner também sugere que devem ser de fácil avaliação. Se houver problemas com o sistema, a equipe de teste de release deve reconhecê-los. Como exemplo de um possível cenário a partir do MHC-PMS, o Quadro 8.1 descreve uma maneira pela qual o sistema pode ser usado em uma visita domiciliar.

O cenário testa uma série de características do MHC-PMS:

1. Autenticação por *logon* no sistema.
2. Download e *upload* de determinados registros do paciente em um computador portátil.
3. Programação de visitas domiciliares.
4. Criptografia e descriptografia de registros de pacientes em um dispositivo móvel.
5. Recuperação e modificação de registros.
6. Links com o banco de dados dos medicamentos, que mantém informações sobre os efeitos colaterais.
7. O sistema de aviso de chamada.

Se você é um testador de release, você percorre esse cenário no papel de Kate, observando como o sistema se comporta em resposta a entradas diferentes. Atuando no papel de Kate, você pode cometer erros deliberados, como introduzir a frase-chave errada para decodificar registros. Esse procedimento verifica a resposta do sistema aos erros. Você deve anotar cuidadosamente quaisquer problemas que possam surgir, incluindo problemas de desempenho. Se um sistema é muito lento, a maneira como ele é usado será diferente. Por exemplo, se leva muito tempo para criptografar um registro, então os usuários que têm pouco tempo podem pular essa etapa. Se eles perderem seu notebook, uma pessoa não autorizada poderia visualizar os registros de pacientes.

Ao usar uma abordagem baseada em cenários, você normalmente testa vários requisitos dentro do mesmo cenário. Assim, além de verificar os requisitos individuais, também está verificando quais combinações de requisitos não causam problemas.

Quadro 8.1 Um cenário de uso para o MHC-PMS

Kate é uma enfermeira especialista em saúde mental. Uma de suas responsabilidades é visitar os pacientes em casa para verificar a eficácia do tratamento e se os pacientes estão sofrendo com os efeitos colaterais da medicação.

Em um dia de visitas, Kate faz o *login* no MHC-PMS e usa-o para imprimir sua agenda de visitas domiciliares para aquele dia, juntamente com o resumo das informações sobre os pacientes a serem visitados. Ela pede que os registros desses pacientes sejam transferidos para seu notebook. É solicitada a palavra-chave para criptografar os registros no notebook.

Um dos pacientes que Kate visita é Jim, que está sendo tratado com medicação para depressão. Jim acha que a medicação está ajudando, mas acredita que tem o efeito colateral de mantê-lo acordado durante a noite. Kate vai consultar o registro de Jim; sua frase-chave é solicitada para decifrar o registro. Ela verifica o medicamento prescrito e consulta seus efeitos colaterais. Insônia é um efeito colateral conhecido. Ela faz uma observação sobre o problema no registro de Jim e sugere que ele visite a clínica para ter sua medicação alterada. Ele concorda. Assim, Kate faz um *prompt* de entrar em contato com ele quando ela voltar à clínica, para que faça uma consulta com um médico. Ela termina a consulta e o sistema criptografa novamente o registro de Jim.

Depois, terminadas suas consultas, Kate retorna à clínica e transfere os registros dos pacientes visitados para o banco de dados. O sistema gera para Kate uma lista de pacientes que ela precisa contatar para obter informações de acompanhamento e fazer agendamentos de consultas.

8.3.3 Testes de desempenho

Uma vez que o sistema tenha sido totalmente integrado, é possível testá-lo para propriedades emergentes, como desempenho e confiabilidade. Os testes de desempenho precisam ser projetados para assegurar que o sistema possa processar a carga a que se destina. Isso normalmente envolve a execução de uma série de testes em que você aumenta a carga até que o desempenho do sistema se torne inaceitável.

Tal como acontece com outros tipos de testes, testes de desempenho estão interessados tanto em demonstrar que o sistema atende seus requisitos quanto em descobrir problemas e defeitos do sistema. Para testar se os requisitos de desempenho estão sendo alcançados, você pode ter de construir um perfil operacional. Um perfil operacional (veja o Capítulo 15) é um conjunto de testes que refletem a mistura real de trabalho que será manipulado pelo sistema. Portanto, se 90% das transações no sistema são do tipo A, 5% são do tipo B e o restante é dos tipos C, D e E, você tem de projetar o perfil operacional de modo que a grande maioria dos testes seja do tipo A. Caso contrário, você não vai ter um teste preciso do desempenho operacional do sistema.

Essa não é necessariamente a melhor abordagem para testes de defeitos. A experiência tem mostrado que uma forma eficaz de descobrir os defeitos é projetar testes para os limites do sistema. Em testes de desempenho, isso significa estressar o sistema, fazendo demandas que estejam fora dos limites de projeto do software. Isso é conhecido como 'teste de estresse'. Por exemplo, digamos que você está testando um sistema de processamento de transações que é projetado para processar até 300 transações por segundo. Você começa a testar esse sistema com menos de 300 transações por segundo; então, aumenta gradualmente a carga no sistema para além de 300 transações por segundo, até que esteja bem além da carga máxima de projeto do sistema e o sistema falhe. Esse tipo de teste tem duas funções:

1. Testar o comportamento de falha do sistema. As circunstâncias podem surgir por meio de uma combinação inesperada de eventos em que a carga sobre o sistema excede a carga máxima prevista. Nessas circunstâncias, é importante que a falha do sistema não cause corrupção de dados ou perda inesperada de serviços de usuário. Testes de estresse que verificam a sobrecarga do sistema fazem com que ele caia de maneira suave em vez de entrar em colapso sob sua carga.
2. Estressar o sistema e trazer à luz defeitos que normalmente não são descobertos. Embora se possa argumentar que esses defeitos, em uso normal, não são suscetíveis a causarem falhas no sistema, pode haver combinações inusitadas de circunstâncias normais que o teste de estresse replique.

Os testes de estresse são particularmente relevantes para sistemas distribuídos baseados em uma rede de processadores. Esses sistemas frequentemente apresentam degradação severa quando estão muito carregados. A rede fica inundada com dados de coordenação que os diferentes processos devem trocar. Os processos tornam-se mais lentos à medida que aguardam os dados requisitados a outros processos. Os testes de estresse ajudam-no a descobrir quando a degradação começa, e, assim, você pode adicionar controles ao sistema para rejeitar operações além desse ponto.

8.4 Testes de usuário

Teste de usuário ou de cliente é um estágio no processo de teste em que os usuários ou clientes fornecem entradas e conselhos sobre o teste de sistema. Isso pode envolver o teste formal de um sistema que foi aprovado por um fornecedor externo ou processo informal em que os usuários experimentam um produto de software novo para ver se gostam e verificar se faz o que eles precisam. O teste de usuário é essencial, mesmo em sistemas abrangentes ou quando testes de release tenham sido realizados. A razão para isso é que as influências do ambiente de trabalho do usuário têm um efeito importante sobre a confiabilidade, o desempenho, a usabilidade e a robustez de um sistema.

É praticamente impossível para um desenvolvedor de sistemas replicar o ambiente de trabalho do sistema, pois os testes no ambiente do desenvolvedor são inevitavelmente artificiais. Por exemplo, um sistema que se destina a ser usado em um hospital é usado em um ambiente clínico em que outras coisas estão acontecendo, como emergências de pacientes, conversas com parentes etc. Isso tudo afeta o uso de um sistema, mas os desenvolvedores não podem incluí-los em seu ambiente de teste.

Na prática, existem três tipos de testes de usuário:

1. Teste alfa, em que os usuários do software trabalham com a equipe de desenvolvimento para testar o software no local do desenvolvedor.
2. Teste beta, em que um release do software é disponibilizado aos usuários para que possam experimentar e levantar os problemas que eles descobriram com os desenvolvedores do sistema.
3. Teste de aceitação, em que os clientes testam um sistema para decidir se está ou não pronto para ser aceito pelos desenvolvedores de sistemas e implantado no ambiente do cliente.

Em testes alfa, usuários e desenvolvedores trabalham em conjunto para testar um sistema que está sendo desenvolvido. Isso significa que os usuários podem identificar os problemas e as questões que não são aparentes para a equipe de testes de desenvolvimento. Os desenvolvedores só podem trabalhar a partir dos requisitos, mas, muitas vezes, estes não refletem outros fatores que afetam o uso prátco do software. Os usuários podem, portanto, fornecer informações sobre as práticas que contribuem com projeto de testes mais realistas.

Os testes alfa são frequentemente usados no desenvolvimento de produtos de software que são vendidos como sistemas-pacote. Os usuários desses produtos podem estar dispostos a se envolver no processo de testes alfa, pois isso lhes antecipa informações sobre novas características do sistema, as quais eles poderão explorar. Também reduz o risco de que mudanças inesperadas no software tenham efeitos perturbadores sobre os negócios. No entanto, testes alfa também podem ser usados quando o software customizado está sendo desenvolvido. Os métodos ágeis, como XP, defendem o envolvimento do usuário no processo de desenvolvimento e alegam que os usuários devem desempenhar um papel fundamental no projeto de testes para o sistema.

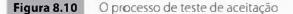
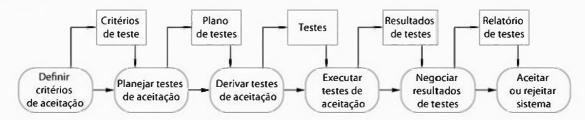
O teste beta ocorre quando um release antecipado, por vezes inacabado, de um sistema de software é disponibilizado aos clientes e usuários para avaliação.

Testadores beta podem ser um grupo de clientes selecionados, os primeiros a adotarem o sistema. Como alternativa, o software pode ser disponibilizado para uso, para qualquer pessoa que esteja interessada nele. O teste beta é usado principalmente para produtos de software que são usados nos mais diversos ambientes (por oposição aos sistemas customizados, que são geralmente usados em um ambiente definido). É impossível para os desenvolvedores de produtos conhecerem e replicarem todos os ambientes em que o software será usado. O teste beta é essencial para descobrir justamente problemas de interação entre o software e as características do ambiente em que ele é usado. Também é uma forma de marketing — os clientes aprendem sobre seu sistema e o que ele pode fazer por eles.

O teste de aceitação é uma parte inerente ao desenvolvimento de sistemas customizados, que ocorre após o teste de release. Engloba o teste formal de um sistema pelo cliente para decidir se ele deve ou não ser aceito. A aceitação designa que o pagamento pelo sistema deve ser feito.

Existem seis estágios no processo de teste de aceitação, como mostra a Figura 8.10. São eles:

1. *Definir critérios de aceitação*. Esse estágio deve, idealmente, ocorrer no início do processo antes de o contrato do sistema ser assinado. Os critérios de aceitação devem ser parte do contrato do sistema e serem acordados entre o cliente e o desenvolvedor. Na prática, porém, pode ser difícil definir critérios para o início do processo. Os requisitos detalhados podem não estar disponíveis e podem haver mudanças significativas dos requisitos durante o processo de desenvolvimento.
2. *Planejar testes de aceitação*. Trata-se de decidir sobre os recursos, tempo e orçamento para os testes de aceitação e estabelecer um cronograma de testes. O plano de teste de aceitação também deve discutir a cobertura dos requisitos exigidos e a ordem em que as características do sistema são testadas. Deve definir os riscos para o processo de testes, como interrupção de sistema e desempenho inadequado, e discutir como esses riscos podem ser mitigados.

Figura 8.10 O processo de teste de aceitação

3. *Derivar testes de aceitação.* Uma vez que os testes de aceitação tenham sido estabelecidos, eles precisam ser projetados para verificar se existe ou não um sistema aceitável. Testes de aceitação devem ter como objetivo testar tanto as características funcionais quanto as não funcionais (por exemplo, desempenho) do sistema. Eles devem, idealmente, fornecer cobertura completa dos requisitos de sistema. Na prática, é difícil estabelecer critérios completamente objetivos de aceitação. Muitas vezes, fica margem para discussão sobre se o teste mostra ou não se um critério foi definitivamente cumprido.
4. *Executar testes de aceitação.* Os testes de aceitação acordados são executados no sistema. Idealmente, isso deve ocorrer no ambiente real em que o sistema será usado, mas isso pode ser interrompido e impraticável. Portanto, um ambiente de testes de usuário pode ter de ser configurado para executar esses testes. É difícil automatizar esse processo, pois parte dos testes de aceitação pode envolver testes de interações entre os usuários finais e o sistema. Pode ser necessário algum treinamento para os usuários finais.
5. *Negociar resultados de teste.* É muito improvável que todos os testes de aceitação definidos passem e que não ocorra qualquer problema com o sistema. Se esse for o caso, então o teste de aceitação está completo, e o sistema pode ser entregue. O mais comum, contudo, é que alguns problemas sejam descobertos. Nesses casos, o desenvolvedor e o cliente precisam negociar para decidir se o sistema é bom o suficiente para ser colocado em uso. Eles também devem concordar sobre a resposta do desenvolvedor para os problemas identificados.
6. *Rejeitar/aceitar sistema.* Esse estágio envolve uma reunião entre os desenvolvedores e o cliente para decidir se o sistema deve ser aceito. Se o sistema não for bom o suficiente para o uso, então será necessário um maior desenvolvimento para corrigir os problemas identificados. Depois de concluída, a fase de testes de aceitação é repetida.

Nos métodos ágeis, como XP, o teste de aceitação tem um significado bastante diferente. Em princípio, ele compartilha a ideia de que os usuários devem decidir quando o sistema é aceitável. No entanto, no XP, o usuário é parte da equipe de desenvolvimento (isto é, ele ou ela é um testador alfa) e fornece os requisitos de sistema em termos de histórias de usuários. Ele ou ela também é responsável pela definição dos testes, que decide se o software desenvolvido apoia ou não a história de usuário. Os testes são automatizados, e o desenvolvimento não continua até os testes de aceitação de história passarem. Portanto, não há uma atividade de teste de aceitação em separado.

Como já discutido no Capítulo 3, um problema com o envolvimento do usuário é garantir que o usuário que está integrado na equipe de desenvolvimento seja um usuário 'típico', com conhecimento geral de como o sistema será usado. Considerando que pode ser difícil encontrar tal usuário, os testes de aceitação não podem refletir verdadeiramente a prática. Além disso, o requisito por testes automatizados limita severamente a flexibilidade de testar sistemas interativos. Para estes sistemas, testes de aceitação podem exigir que grupos de usuários finais usem o sistema como se fosse parte de seu trabalho diário.

Você pode pensar que o teste de aceitação é uma questão clara de corte contratual. Se um sistema não passar em seus testes de aceitação, então não deve ser aceito, e o pagamento não deve ser feito. No entanto, a realidade é mais complexa. Os clientes querem usar o software assim que possível por causa dos benefícios de sua implantação imediata. Eles podem ter comprado novos equipamentos, treinado seu pessoal e alterado seus processos. Eles podem estar dispostos a aceitar o software, independentemente dos problemas, porque os custos do não uso do software são maiores do que os custos de se trabalhar nos problemas. Portanto, o resultado das negociações pode ser de aceitação condicional do sistema. O cliente pode aceitar o sistema para que a implantação possa começar. E o fornecedor do sistema se compromete a sanar os problemas urgentes e entregar uma nova versão para o cliente o mais rápido possível.

PONTOS IMPORTANTES

- Os testes só podem anunciar a presença de defeitos em um programa. Não podem demonstrar que não existem defeitos remanescentes.
- Testes de desenvolvimento são de responsabilidade da equipe de desenvolvimento de software. Outra equipe deve ser responsável por testar o sistema antes que ele seja liberado para os clientes. No processo de testes de usuário, clientes ou usuários do sistema fornecem dados de teste e verificam se os testes são bem-sucedidos.
- Testes de desenvolvimento incluem testes unitários, nos quais você testa objetos e métodos específicos; testes de componentes, em que você testa diversos grupos de objetos; e testes de sistema, nos quais você testa sistemas parciais ou completos.
- Ao testar o software, você deve tentar 'quebrar' o software usando sua experiência e diretrizes para escolher os tipos de casos de teste que têm sido eficazes na descoberta de defeitos em outros sistemas.

- Sempre que possível, você deve escrever testes automatizados. Os testes são incorporados em um programa que pode ser executado cada vez que uma alteração é feita para um sistema.
- O desenvolvimento *test-first* é uma abordagem de desenvolvimento na qual os testes são escritos antes do código que será testado. Pequenas alterações no código são feitas, e o código é refatorado até que todos os testes sejam executados com êxito.
- Testes de cenário são úteis porque replicam o uso prático do sistema. Trata-se de inventar um cenário típico de uso e usar isso para derivar casos de teste.
- Teste de aceitação é um processo de teste de usuário no qual o objetivo é decidir se o software é bom o suficiente para ser implantado e usado em seu ambiente operacional.

LEITURA COMPLEMENTAR

'How to design practical test cases'. Um artigo sobre como projetar casos de teste, escrito por um autor de uma empresa japonesa que tem uma reputação muito boa em entregar softwares com pouquíssimos defeitos. (YAMAURA, T. *IEEE Software*, v. 15, n. 6, nov. 1998. Disponível em: <http://dx.doi.org/10.1109/52.730835>.)

How to Break Software: A Practical Guide to Testing. Esse é um livro sobre testes de software, mais prático do que teórico, no qual o autor apresenta um conjunto de diretrizes baseadas em experiências em projetar testes suscetíveis de serem eficazes na descoberta de defeitos de sistema. (WHITTAKER, J. A. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley, 2002.)

'Software Testing and Verification'. Essa edição especial do *IBM Systems Journal* abrange uma série de artigos sobre testes, incluindo uma boa visão geral, artigos sobre métricas e automação de teste. (*IBM Systems Journal*, v. 41, n. 1, jan. 2002.)

'Test-Driven Development'. Essa edição especial sobre desenvolvimento dirigido a testes inclui uma boa visão geral de TDD, bem como artigos de experiência em como TDD tem sido usado para diferentes tipos de software. (*IEEE Software*, v. 24, n. 3, mai./jun. 2007.)

EXERCÍCIOS

8.1 Explique por que um programa não precisa, necessariamente, ser completamente livre de defeitos antes de ser entregue a seus clientes.

8.2 Explique por que os testes podem detectar apenas a presença de erros, e não sua ausência.

8.3 Algumas pessoas argumentam que os desenvolvedores não devem ser envolvidos nos testes de seu próprio código, mas que todos os testes devem ser de responsabilidade de uma equipe independente. Dê argumentos a favor e contra a realização de testes pelos próprios desenvolvedores.

8.4 Você foi convidado para testar um método chamado 'catWhiteSpace' em um objeto 'Parágrafo'. Dentro do parágrafo, as sequências de caracteres em branco são substituídas por um único caractere em branco. Identifique partições para testar esse exemplo e derive um conjunto de testes para o método 'catWhiteSpace'.

8.5 O que é o teste de regressão? Explique como o uso de testes automatizados e um *framework* de teste, tal qual o JUnit, simplifica os testes de regressão.

8.6 O MHC-PMS é construído por meio da adaptação de um sistema de informações de prateleira. Quais são as diferenças entre esses testes de software e os testes de um sistema desenvolvido por meio de uma linguagem orientada a objetos como Java?

8.7 Escreva um cenário que poderia ser usado para ajudar no projeto de testes para o sistema da estação meteorológica no deserto.

8.8 O que você entende pelo termo 'testes de estresse'? Sugira como você pode estressar o teste MHC-PMS.

8.9 Quais são as vantagens de os usuários se envolverem nos testes de release em um estágio inicial do processo de teste? Existem desvantagens na participação do usuário?

8.10 Uma abordagem comum para o teste de sistema é testar o sistema até que o orçamento de teste esteja esgotado e, em seguida, entregar o sistema para os clientes. Discuta a ética dessa abordagem para os sistemas que são entregues aos clientes externos.

REFERÊNCIAS

ANDREA, J. Envisioning the Next Generation of Functional Testing Tools. *IEEE Software*, v. 24, n. 3, 2007, p. 58-65.

BECK, K. *Test Driven Development*: By Example. Boston: Addison-Wesley, 2002.

BEZIER, B. *Software Testing Techniques*. 2. ed. Nova York: Van Nostrand Rheinhold, 1990.

BOEHM, B. W. Software engineering; R & D Trends and defense needs. In: *Research Directions in Software Technology*. WEGNER, P. (Org.). Cambridge, Mass.: MIT Press, 1979, p. 1-9.

CUSAMANO, M.; SELBY, R. W. *Microsoft Secrets*. Nova York: Simon and Shuster, 1998.

DIJKSTRA, E. W.; DAHL, O. J.; HOARE, C. A. R. *Structured Programming*. Londres: Academic Press, 1972.

FAGAN, M. E. Advances in Software Inspections. *IEEE Trans. on Software Eng*, v. SE-12, n. 7, 1986, p. 744-751.

JEFFRIES, R.; MELNIK, G. TDD: The Art of Fearless Programming. *IEEE Software*, v. 24, 2007, p. 24-30.

KANER, C. The power of 'What If...' and nine ways to fuel your imagination: Cem Kaner on scenario testing. *Software Testing and Quality Engineering*, v. 5, n. 5, 2003, p. 16-22.

LUTZ, R. R. Analyzing Software Requirements Errors in Safety-Critical Embedded Systems. *RE'93*, San Diego, Calif.: IEEE, 1993.

MARTIN, R. C. Professionalism and Test-Driven Development. *IEEE Software*, v. 24, n. 3, 2007, p. 32-36.

MASSOL, V.; HUSTED, T. *JUnit in Action*. Greenwich, Conn.: Manning Publications Co., 2003.

PROWELL, S. J.; TRAMMELL, C. J.; Linger, R. C.; POORE, J. H. *Cleanroom Software Engineering*: Technology and Process. Reading, Mass.: Addison-Wesley, 1999.

WHITTAKER, J. W. *How to Break Software*: A Practical Guide to Testing. Boston: Addison-Wesley, 2002.

CAPÍTULO

1 2 3 4 5 6 7 8 **9** 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Evolução de software

Objetivos

Os objetivos deste capítulo são explicar por que a evolução é uma parte importante da engenharia de software e descrever os processos de evolução de software. Com a leitura deste capítulo, você:

- compreenderá que a mudança é inevitável caso os sistemas de software devam permanecer úteis, e que o desenvolvimento e a evolução de software podem ser integrados em um modelo em espiral;
- compreenderá os processos de evolução de software e as influências sobre esses processos;
- terá aprendido os diferentes tipos de manutenção de software e os fatores que afetam seus custos;
- entenderá como os sistemas legados podem ser avaliados para decidir se devem ser descartados, mantidos, passar por reengenharia ou substituídos.

Conteúdo

O desenvolvimento de software não é interrompido quando o sistema é entregue, mas continua por toda a vida útil do sistema. Depois que o sistema é implantado, para que ele se mantenha útil é inevitável que ocorram mudanças —, mudanças nos negócios e nas expectativas dos usuários, que geram novos requisitos para o software. Partes do software podem precisar ser modificadas para corrigir erros encontrados na operação, para que o software se adapte às alterações de sua plataforma de hardware e software, bem como para melhorar seu desempenho ou outras características não funcionais.

A evolução do software é importante, pois as organizações investem grandes quantias de dinheiro em seus softwares e são totalmente dependentes desses sistemas. Seus sistemas são ativos críticos de negócios, e as organizações devem investir nas mudanças de sistemas para manter o valor desses ativos. Consequentemente, a maioria das grandes empresas gasta mais na manutenção de sistemas existentes do que no desenvolvimento de novos sistemas. Baseado em uma pesquisa informal da indústria, Erlikh (2000) sugere que 85% a 90% dos custos organizacionais de software são custos de evolução. Outras pesquisas sugerem que cerca de dois terços de custos de software são de evolução. Com certeza, os custos para mudança de software requerem grande parte do orçamento de TI para todas as empresas.

Uma evolução de software pode ser desencadeada por necessidades empresariais em constante mudança, por relatos de defeitos de software ou por alterações de outros sistemas em um ambiente de software. Hopkins e Jenkins (2008) cunharam o termo 'desenvolvimento de software *brownfield*' para descrever situações nas quais os sistemas de

software precisam ser desenvolvidos e gerenciados em um ambiente em que dependem de vários outros sistemas de software.

Portanto, a evolução de um sistema raramente pode ser considerada de forma isolada. Alterações no ambiente levam a mudanças nos sistemas que podem, então, provocar mais mudanças ambientais. Certamente, o fato de os sistemas terem de evoluir em um 'ambiente rico' costuma aumentar as dificuldades e os custos de evolução. Assim como a compreensão e análise do impacto de uma mudança proposta no sistema em si, você também pode avaliar como isso pode afetar outros sistemas operacionais do ambiente.

Sistemas de software úteis muitas vezes têm uma vida útil muito longa. Por exemplo, sistemas militares ou de infraestrutura de grande porte, como sistemas de controle de tráfego aéreo, podem ter uma vida de 30 anos ou mais. Sistemas de negócios têm, usualmente, mais de dez anos de idade. Como os softwares custam muito dinheiro, uma empresa pode usar um sistema por muitos anos para ter retorno de seu investimento. É claro que os requisitos dos sistemas instalados mudam de acordo com os negócios e suas mudanças de ambiente. Portanto, os novos *releases* dos sistemas, que incorporam as alterações e atualizações, são geralmente criados em intervalos regulares.

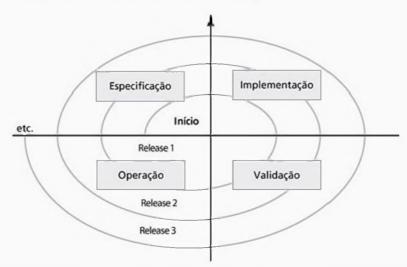
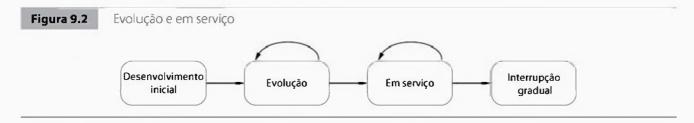
Portanto, você deve pensar na engenharia de software como um processo em espiral com requisitos, projeto, implementação e testes que dura toda a vida útil do sistema (Figura 9.1). Você começa criando o *release* 1 do sistema. Uma vez entregue, alterações são propostas, e o desenvolvimento do *release* 2 começa imediatamente. De fato, a necessidade de evolução pode se tornar óbvia mesmo antes de o sistema ser implantado, de modo que os *releases* posteriores do software possam ser desenvolvidos antes de o *release* atual ser lançado.

Esse modelo de evolução do software implica que uma única organização é responsável tanto pelo desenvolvimento de software inicial quanto por sua evolução. A maioria dos pacotes de software é desenvolvida com essa abordagem. Para softwares customizados, uma abordagem diferente costuma ser usada. Uma empresa de software desenvolve softwares para um cliente, e ela possui uma equipe própria de desenvolvimento que assume o sistema. Ela é responsável pela evolução do software. Como alternativa, o cliente pode emitir um contrato separado para outra empresa cuidar do suporte e da evolução do sistema.

Nesse caso, as descontinuidades no processo em espiral são bem possíveis. Documentos de requisitos e de projeto não podem ser passados de uma empresa para outra. Empresas podem fundir-se ou reorganizar-se e herdar software de outras empresas e, em seguida, perceber que o software deve ser mudado. Quando a transição do desenvolvimento para a evolução é imperceptível, o processo de mudança do software após a entrega é, muitas vezes, chamado 'manutenção de software'. Como discuto adiante, a manutenção envolve atividades adicionais de processo, como o entendimento de programa, além das atividades normais de desenvolvimento de software.

Como mostra a Figura 9.2, Rajlich e Bennett (2000) propuseram uma visão alternativa do ciclo de vida de evolução do software. Nesse modelo, eles distinguem 'evolução' e 'em serviço'. A evolução é a fase em que mudanças significativas na arquitetura e funcionalidade do software podem ser feitas. Quando em serviço, as únicas mudanças feitas são relativamente pequenas, essenciais.

Figura 9.1 Um modelo em espiral de desenvolvimento e evolução

Figura 9.2 Evolução e em serviço

Durante a evolução, o software é usado com sucesso, e existe um fluxo constante de propostas de alterações de requisitos. No entanto, como o software é modificado, sua estrutura tende a degradar e as mudanças ficam mais e mais caras. Isso ocorre com frequência depois de alguns anos de uso, quando outras mudanças ambientais, como o hardware e sistemas operacionais, também são necessárias. Em algum estágio do ciclo de vida, o software chega a um ponto de transição em que mudanças significativas e implementação de novos requisitos se tornam menos rentáveis.

Nessa fase, o software passa da evolução para o serviço. Durante o serviço, o software ainda é útil e usado, mas apenas pequenas mudanças táticas são feitas. Normalmente, durante essa fase, a empresa está considerando como o software pode ser substituído. Na fase final, na interrupção gradual, o software ainda pode ser usado, mas não são implementadas novas mudanças. Os usuários precisam contornar qualquer problema que descubram.

9.1 Processos de evolução

A evolução dos processos de software pode variar dependendo do tipo de software que esteja sendo mantido, dos processos de desenvolvimento usados em uma organização e das habilidades das pessoas envolvidas. Em algumas organizações, a evolução pode ser um processo informal em que as solicitações de mudança resultam, na maior parte, das conversas entre os usuários do sistema e desenvolvedores. Em outras empresas, é um processo formal com documentação estruturada produzida em cada estágio do processo.

Em todas as organizações, as propostas de mudança no sistema são os acionadores para a evolução. As propostas de mudança podem vir de requisitos já existentes que não tenham sido implementados no *release* de sistema, solicitações de novos requisitos, relatórios de *bugs* do sistema apontados pelos *stakeholders* e novas ideias para melhoria do software vindas da equipe de desenvolvimento. Os processos de identificação de mudanças e de evolução de sistema são cíclicos e continuam durante toda a vida de um sistema (Figura 9.3).

As propostas de mudança devem ser relacionadas aos componentes do sistema que necessitam ser modificados para implementar essas propostas, o que permite que o custo e o impacto da alteração sejam avaliados. Isso faz parte do processo geral de gerenciamento de mudanças, que também deve assegurar que as versões corretas dos componentes estejam incluídas em cada *release* do sistema. No Capítulo 25, discuto gerenciamento de mudanças e de configuração.

Figura 9.3 Processos de identificação de mudanças e de evolução

A Figura 9.4, adaptada de Arthur (1988), mostra uma visão geral do processo de evolução. O processo inclui as atividades fundamentais de análise de impacto, planejamento de *release*, implementação de sistema e liberação de um sistema para os clientes. O custo e o impacto dessas mudanças são avaliados para ver quanto do sistema é afetado pelas mudanças e quanto poderia custar para implementá-las. Se as mudanças propostas são aceitas, um novo *release* do sistema é planejado. Durante o planejamento de *release*, todas as mudanças propostas (correção de defeitos, adaptação e nova funcionalidade) são consideradas. Uma decisão é tomada de acordo com as mudanças a serem implementadas na próxima versão do sistema. As mudanças são implementadas e validadas, e uma nova versão do sistema é liberada. O processo itera com um novo conjunto de mudanças propostas para o próximo *release*.

Você pode pensar em implementação de mudanças como uma iteração do desenvolvimento, em que as revisões do sistema são projetadas, implementadas e testadas. No entanto, uma diferença fundamental é que o primeiro estágio da implementação da mudança pode envolver a compreensão do programa, especialmente se os desenvolvedores do sistema original não forem responsáveis pela implementação de mudanças. Durante esse estágio de compreensão do programa, é necessário que se entenda como o programa está estruturado, como implementa a funcionalidade e como a mudança proposta pode afetá-lo. Você precisa desse entendimento para se certificar de que as mudanças implementadas não causarão novos problemas quando forem implementadas em um sistema existente.

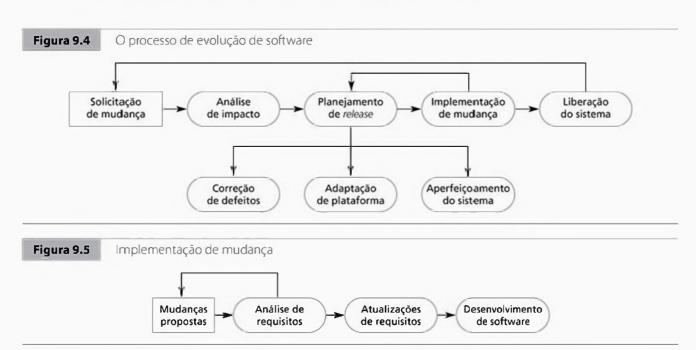
Idealmente, o estágio de implementação desse processo deve modificar a especificação, o projeto e a implementação do sistema para refletir as alterações do sistema (Figura 9.5). Novos requisitos que refletem as mudanças do sistema são propostos, analisados e validados. Componentes do sistema são reprojetados e implementados, e o sistema é testado novamente. Se for o caso, a prototipação das alterações propostas pode ser realizada como parte do processo de análise de mudanças.

Durante o processo de evolução, os requisitos são analisados detalhadamente, e as implicações das mudanças surgem onde não eram aparentes no processo anterior de análise. Isso significa que as alterações propostas podem ser modificadas e mais discussões com os clientes podem ser necessárias antes que estas sejam implementadas.

Às vezes, as solicitações de mudança são relacionadas a problemas do sistema que devem ser resolvidos com urgência. Essas mudanças urgentes podem surgir por três motivos:

1. Se ocorrer um defeito grave no sistema, que precisa ser corrigido para permitir a continuidade do funcionamento normal.
2. Se as alterações no ambiente operacional dos sistemas tiverem efeitos inesperados que interrompam o funcionamento normal.
3. Se houver mudanças inesperadas no funcionamento do negócio que executa o sistema, como o surgimento de novos concorrentes ou a introdução de nova legislação que afete o sistema.

Figura 9.4 O processo de evolução de software

Figura 9.5 Implementação de mudança

Nesses casos, a necessidade de fazer a mudança rapidamente significa que você pode não ser capaz de acompanhar o processo de análise formal da mudança. Em vez de modificar os requisitos e o projeto, para resolver o problema imediatamente você faz uma correção de emergência no programa (Figura 9.6). No entanto, o perigo é que os requisitos, o projeto do software e o código podem tornar-se inconsistentes. Embora você possa ter a intenção de documentar a mudança nos requisitos e no projeto, correções adicionais de emergência para o software podem, então, ser necessárias. Elas têm prioridade sobre a documentação. Eventualmente, a mudança original é esquecida e a documentação e o código do sistema nunca são realinhados.

Em geral, as correções de emergência no sistema devem ser concluídas o mais rapidamente possível. Considerando a estrutura do sistema, você possivelmente optou por uma solução rápida e viável, e não a melhor solução. Isso acelera o processo de envelhecimento de software, faz que futuras alterações se tornem progressivamente mais difíceis, e os custos de manutenção, mais altos.

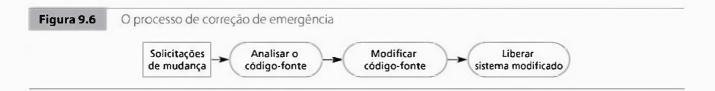
Idealmente, quando são feitas correções de emergência no código, após os defeitos serem corrigidos, a solicitação de mudança deve permanecer em aberto. O código de emergência pode então ser reimplementado mais cuidadosamente após uma análise mais aprofundada. Certamente, o código usado na correção pode ser reusado. Uma alternativa, uma melhor solução para o problema pode ser descoberta quando houver mais tempo disponível para análise. Na prática, porém, é quase inevitável que essas melhorias tenham baixa prioridade. Frequentemente, elas são esquecidas, sobretudo se as alterações são feitas no sistema; então, torna-se muito complicado refazer as correções de emergência.

Métodos e processos ágeis discutidos no Capítulo 3 podem ser usados para a evolução de programa, bem como para seu desenvolvimento. Na verdade, por esses métodos serem baseados em desenvolvimento incremental, a transição de desenvolvimento ágil para evolução pós-entrega deveria ser imperceptível. Técnicas como os testes de regressão automatizados são úteis quando são feitas alterações no sistema. As alterações podem ser expressas como histórias de usuário e o envolvimento do cliente pode priorizar as mudanças necessárias em um sistema funcional. Em suma, simplesmente, a evolução envolve a continuação do processo de desenvolvimento ágil.

No entanto, podem surgir problemas em situações em que ocorre a transferência de uma equipe de desenvolvimento para uma equipe independente responsável pela evolução. Existem duas situações possivelmente problemáticas:

1. Quando a equipe de desenvolvimento usou uma abordagem ágil, mas a equipe de evolução não está familiarizada com os métodos ágeis e prefere uma abordagem baseada em planos. A equipe de evolução pode esperar uma documentação detalhada para apoiar a evolução, e, em processos ágeis, esta raramente é produzida. Não pode haver qualquer declaração definitiva de requisitos do sistema que possa ser modificada enquanto as alterações no sistema são feitas.
2. Quando uma abordagem baseada em planos for usada para o desenvolvimento, mas a equipe de evolução preferir usar métodos ágeis. Nesse caso, a equipe de evolução pode ter de começar do zero a partir do desenvolvimento de testes automatizados, e os códigos do sistema podem não ser refatorados e simplificados, como se prevê no desenvolvimento ágil. Nesse caso, alguma reengenharia pode ser necessária para melhorar o código antes que ele possa ser usado em um processo de desenvolvimento ágil.

Poole e Huisman (2001) relatam suas experiências no uso de 'Extreme Programming' para a manutenção de um sistema de grande porte originalmente desenvolvido pela abordagem baseada em planos. Depois da reengenharia do sistema para melhoria da sua estrutura, o XP foi usado com êxito no processo de manutenção.

Figura 9.6 O processo de correção de emergência

9.2 Dinâmica da evolução de programas

A dinâmica da evolução de programas é o estudo da mudança de sistema. Nas décadas de 1970 e 1980, Lehman e Belady (1985) realizaram vários estudos empíricos sobre a mudança de sistema com intenção de compreender mais sobre as características de evolução de software. O trabalho continuou na década de 1990 com Lehman e outros pesquisando o significado de *feedback* nos processos de evolução (LEHMAN, 1996; LEHMAN et al., 1998; LEHMAN et al., 2001). A partir desses estudos, eles propuseram as 'Leis de Lehman', relativas às mudanças de sistema (Tabela 9.1).

Lehman e Belady alegam que essas leis são suscetíveis de serem verdadeiras para todos os tipos de sistemas de software de grandes organizações (que eles chamam sistemas E-type). Nesses sistemas, os requisitos estão mudando para refletir as necessidades dos negócios. Novos *releases* do sistema são essenciais para o sistema fornecer valor ao negócio.

A primeira lei afirma que a manutenção de sistema é um processo inevitável. Como o ambiente do sistema muda, novos requisitos surgem, e o sistema deve ser modificado. Quando o sistema modificado é reintroduzido no ambiente, este promove mais mudanças no ambiente, de modo que o processo de evolução recomeça.

A segunda lei afirma que, quando um sistema é alterado, sua estrutura se degrada. A única maneira de evitar que isso aconteça é investir em manutenção preventiva. Você gasta tempo melhorando a estrutura do software sem aperfeiçoar sua funcionalidade. Obviamente, isso implica custos adicionais, além da implementação das mudanças de sistema exigidas.

A terceira lei é, talvez, a mais interessante e a mais controversa das leis de Lehman. Ela sugere que sistemas de grande porte têm uma dinâmica própria, estabelecida em um estágio inicial do processo de desenvolvimento. Isso determina a tendência geral do processo de manutenção de sistema e limita o número de possíveis alterações no mesmo. Lehman e Belady sugerem que essa lei seja uma consequência de fatores estruturais que influenciam e restringem a mudança do sistema, bem como os fatores organizacionais que afetam o processo de evolução.

Os fatores estruturais que afetam a terceira lei resultam da complexidade dos sistemas de grande porte. Assim que você altera e amplia um programa, sua estrutura tende a degradar. Isso é verdadeiro para todos os tipos de sistemas (não apenas software) e ocorre porque você está adaptando uma estrutura destinada a uma finalidade para uma finalidade diferente. Se essa degradação não for verificada, torna-se mais e mais difícil fazer alterações

Tabela 9.1 Leis de Lehman

Lei	Descrição
Mudança contínua	Um programa usado em um ambiente do mundo real deve necessariamente mudar, ou se torna progressivamente menos útil nesse ambiente.
Aumento da complexidade	Como um programa em evolução muda, sua estrutura tende a tornar-se mais complexa. Recursos extras devem ser dedicados a preservar e simplicar a estrutura.
Evolução de programa de grande porte	A evolução de programa é um processo de autorregulação. Atributos de sistema como tamanho, tempo entre *releases* e número de erros relatados são aproximadamente invariáveis para cada *release* do sistema.
Estabilidade organizacional	Ao longo da vida de um programa, sua taxa de desenvolvimento é aproximadamente constante e independente dos recursos destinados ao desenvolvimento do sistema.
Conservação da familiaridade	Durante a vigência de um sistema, a mudança incremental em cada *release* é aproximadamente constante.
Crescimento contínuo	A funcionalidade oferecida pelos sistemas tem de aumentar continuamente para manter a satisfação do usuário.
Declínio de qualidade	A qualidade dos sistemas cairá, a menos que eles sejam modificados para refletir mudanças em seu ambiente operacional.
Sistema de *feedback*	Os processos de evolução incorporam sistemas de *feedback* multiagentes, *multiloop*, e você deve tratá-los como sistemas de *feedback* para alcançar significativa melhoria do produto.

no programa. Fazer pequenas alterações diminui o grau de degradação estrutural e, assim, diminui os riscos que causam sérios problemas de confiança do sistema. Se você fizer grandes alterações, existe alta probabilidade de essas alterações introduzirem novos defeitos, os quais, por sua vez, inibem outras mudanças no programa.

Os fatores organizacionais que afetam a terceira lei refletem o fato de os sistemas de grande porte geralmente serem produzidos por grandes empresas. Essas empresas têm burocracias internas que definem o orçamento de mudanças para cada sistema e controlam o processo de tomada de decisão. As empresas têm de tomar decisões sobre os riscos e o valor das alterações e sobre os custos envolvidos. Tais decisões levam tempo para serem tomadas e, às vezes, leva-se mais tempo para decidir sobre as alterações a serem feitas do que para implementá-las. A velocidade do processo decisório da organização, portanto, regula a taxa de mudança do sistema.

A quarta lei de Lehman sugere que a maioria dos grandes projetos de programação trabalha em um estado 'saturado'. Ou seja, uma mudança de recursos ou de pessoal tem efeitos imperceptíveis sobre a evolução do sistema a longo prazo. Isso é consistente com a terceira lei, que sugere que a evolução de programa é independente das decisões de gerenciamento. Essa lei confirma que grandes equipes de desenvolvimento de software são, em muitos casos, improdutivas, porque os *overheads* de comunicação dominam o trabalho da equipe.

A quinta lei de Lehman está preocupada com o aumento das mudanças em cada *release* de sistema. Adicionar nova funcionalidade a um sistema inevitavelmente introduz novos defeitos. Quanto mais funcionalidade for adicionada em cada versão, mais falhas haverá. Portanto, uma grande adição de funcionalidade em um *release* de sistema significa que este deverá ser seguido por um *release* posterior, no qual os defeitos do novo sistema serão corrigidos e uma nova funcionalidade relativamente pequena deverá ser incluída. Essa lei sugere que não deve haver orçamento para grandes incrementos de funcionalidade em cada *release* sem levar em conta a necessidade de correção de defeitos.

As cinco primeiras leis estavam nas propostas iniciais de Lehman; as leis remanescentes foram adicionadas posteriormente, em outro trabalho. A sexta e sétima leis são semelhantes e, essencialmente, dizem que os usuários de um software estarão a cada dia mais descontentes com ele, a menos que ele seja mantido e que nova funcionalidade seja adicionada. A última lei reflete o mais recente trabalho com processos de *feedback*, embora ainda não esteja claro como isso pode ser aplicado em desenvolvimento prático de software.

As observações de Lehman parecem, de modo geral, sensatas. Devem ser levadas em consideração ao se planejar o processo de manutenção. É possível que, em alguns momentos, considerações comerciais possam nos forçar a ignorá-las. Por exemplo, por razões de marketing, pode ser necessário fazer várias alterações no sistema principal em um único *release*. As prováveis consequências disso são a necessidade de um ou mais *releases* dedicados à correção de erros. Muitas vezes, você vê isso em softwares de computadores pessoais, quando um importante novo *release* de uma aplicação é seguido rapidamente por uma atualização de correção de *bugs*.

9.3 Manutenção de software

A manutenção de software é o processo geral de mudança em um sistema depois que ele é liberado para uso. O termo geralmente se aplica ao software customizado em que grupos de desenvolvimento separados estão envolvidos antes e depois da liberação. As alterações feitas no software podem ser simples mudanças para correção de erros de codificação, até mudanças mais extensas para correção de erros de projeto, ou melhorias significativas para corrigir erros de especificação ou acomodar novos requisitos. As mudanças são implementadas por meio da modificação de componentes do sistema existente e, quando necessário, por meio da adição de novos componentes.

Existem três diferentes tipos de manutenção de software:

1. *Correção de defeitos.* Erros de codificação são relativamente baratos para serem corrigidos; erros de projeto são mais caros, pois podem implicar reescrever vários componentes de programa. Erros de requisitos são os mais caros para se corrigir devido ao extenso reprojeto de sistema que pode ser necessário.
2. *Adaptação ambiental.* Esse tipo de manutenção é necessário quando algum aspecto do ambiente do sistema, como o hardware, a plataforma do sistema operacional ou outro software de apoio sofre uma mudança. O sistema de aplicação deve ser modificado para se adaptar a essas mudanças de ambiente.
3. *Adição de funcionalidade.* Esse tipo de manutenção é necessário quando os requisitos de sistema mudam em resposta às mudanças organizacionais ou de negócios. A escala de mudanças necessárias para o software é, frequentemente, muito maior do que para os outros tipos de manutenção.

Na prática, não existe uma distinção clara entre esses tipos de manutenção. Ao adaptar o sistema a um novo ambiente, você pode adicionar funcionalidade para tirar proveito de novas características do ambiente. Os defeitos de software são frequentemente expostos porque os usuários usam o sistema de formas inesperadas. Mudar o sistema para acomodar sua maneira de trabalhar é a melhor maneira de corrigir tais defeitos.

Esses tipos de manutenção geralmente são reconhecidos, mas pessoas costumam dar-lhes nomes diferentes. 'Manutenção corretiva' é universalmente usado para se referir à manutenção para corrigir defeitos. No entanto, 'manutenção adaptativa' significa, em alguns casos, adaptação ao novo ambiente e, em outros, adaptar o software aos novos requisitos. 'Manutenção perfectiva' às vezes significa aperfeiçoar o software por meio da implementação de novos requisitos; às vezes, significa manutenção de funcionalidade de sistema para melhorar sua estrutura e seu desempenho. Em razão dessa incerteza quanto à nomenclatura, neste capítulo tenho evitado o uso de todos esses termos.

Existem vários estudos de manutenção de software que observaram os relacionamentos entre a manutenção e o desenvolvimento e entre as diferentes atividades de manutenção (KROGSTIE et al., 2005;. LIENTZ e SWANSON, 1980; NOSEK e PALVIA, 1990; SOUSA, 1998). Devido às diferenças de terminologia, os detalhes desses estudos não podem ser comparados. Apesar das mudanças na tecnologia e dos diferentes domínios de aplicação, parece ter havido pouquíssimas mudanças na distribuição do esforço de evolução desde a década de 1980.

As pesquisas em geral concordam que a manutenção de software ocupa uma proporção maior dos orçamentos de TI que o desenvolvimento (a manutenção detém, aproximadamente, dois terços do orçamento, contra um terço para desenvolvimento). Elas também concordam que se gasta mais do orçamento de manutenção na implementação de novos requisitos do que na correção de *bugs*. A Figura 9.7 mostra, aproximadamente, a distribuição dos custos de manutenção. As porcentagens específicas variam de uma organização para outra, mas, universalmente, a correção de defeitos de sistema não é a atividade de manutenção mais cara. Evoluir o sistema para lidar com novos ambientes e novos ou alterados requisitos consome mais esforço de manutenção.

Os custos relativos de manutenção e desenvolvimento variam de um domínio de aplicação para outro. Guimaraes (1983) verificou que os custos de manutenção para sistemas de aplicação de negócio são comparáveis aos custos de desenvolvimento de sistema. Para sistemas embutidos de tempo real, os custos de manutenção são até quatro vezes maiores do que os custos de desenvolvimento. A alta confiabilidade e os requisitos de desempenho desses sistemas significam que os módulos devem ser fortemente ligados e, portanto, difíceis de serem alterados. Ainda que essas estimativas tenham mais de 25 anos, é improvável que a distribuição de custos para diferentes tipos de sistema tenha mudado significativamente.

Geralmente, vale a pena investir esforços no projeto e implementação de um sistema para redução de custos de mudanças futuras. Adicionar uma nova funcionalidade após a liberação é caro porque é necessário tempo para aprender o sistema e analisar o impacto das alterações propostas. Portanto, o trabalho feito durante o desenvolvimento para tornar a compreensão e a mudança no software mais fáceis provavelmente reduzirá os custos de evolução. Boas técnicas de engenharia de software, como descrição precisa, uso de desenvolvimento orientado a objetos e gerenciamento de configuração, contribuem para a redução dos custos de manutenção.

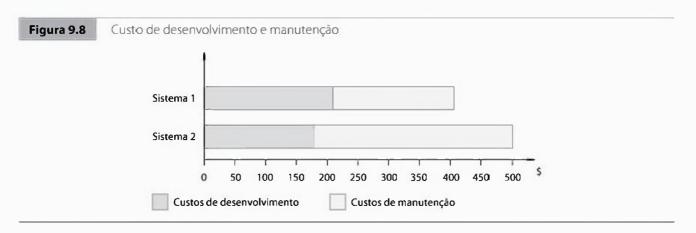
Figura 9.7 Distribuição do esforço de manutenção

A Figura 9.8 mostra como maiores esforços durante o desenvolvimento do sistema para produção de um sistema manutenível reduzem os custos gerais durante a vida útil do sistema. Devido à redução potencial de custos de compreensão, análise e testes, existe um significativo efeito multiplicador quando o sistema é desenvolvido para manutenção. Para Sistema 1, o custo extra de desenvolvimento de 25 mil dólares é investido para tornar o sistema mais manutenível. Isso resulta em uma economia de cem mil dólares em custos de manutenção durante a vida útil do sistema, o que pressupõe que um aumento percentual no custo de desenvolvimento resulta na redução dos custos gerais do sistema em uma porcentagem comparável.

Essas estimativas são hipotéticas, mas não há dúvida de que o desenvolvimento de software para torná-lo mais manutenível é viável, quando todos os custos da vida do software são levados em conta. Essa é a razão da refatoração em desenvolvimento ágil. Sem refatoração, as mudanças no código tornam-se cada vez mais difíceis e caras. No entanto, no desenvolvimento dirigido a planos, a realidade é que raramente são feitos investimentos adicionais na melhoria do código durante o desenvolvimento. Isso é devido, principalmente, às maneiras como as organizações lidam com seus orçamentos. Investir na manutenção leva a aumentos de custos a curto prazo, que são mensuráveis. Infelizmente, os ganhos de longo prazo não podem ser medidos, assim como as empresas são relutantes em gastar dinheiro em um retorno futuro incerto.

Geralmente, é mais caro adicionar funcionalidade depois que um sistema está em operação do que implementar a mesma funcionalidade durante o desenvolvimento. As razões para isso são:

1. *Estabilidade da equipe*. Depois de um sistema ter sido liberado, é normal que a equipe de desenvolvimento seja desmobilizada e as pessoas sejam remanejadas para novos projetos. A nova equipe ou as pessoas responsáveis pela manutenção do sistema não entendem o sistema ou não entendem a estrutura para tomar as decisões de projeto. Antes de implementar alterações é preciso investir tempo em compreender o sistema existente.
2. *Más práticas de desenvolvimento*. O contrato para a manutenção de um sistema é geralmente separado do contrato de desenvolvimento do sistema. O contrato de manutenção pode ser dado a uma empresa diferente da do desenvolvedor do sistema original. Esse fator, juntamente com a falta de estabilidade da equipe, significa que não há incentivo para a equipe de desenvolvimento escrever um software manutenível. Se uma equipe de desenvolvimento pode cortar custos para poupar esforço durante o desenvolvimento, vale a pena fazê-lo, mesmo que isso signifique que o software será mais difícil de mudar no futuro.
3. *Qualificações de pessoal*. A equipe de manutenção é relativamente inexperiente e não familiarizada com o domínio de aplicação. A manutenção tem uma imagem pobre entre os engenheiros de software. É vista como um processo menos qualificado do que o desenvolvimento de sistema e é muitas vezes atribuída ao pessoal mais jovem. Além disso, os sistemas antigos podem ser escritos em linguagens obsoletas de programação. A equipe de manutenção pode não ter muita experiência de desenvolvimento nessas linguagens e precisa primeiro aprender para depois manter o sistema.
4. *Idade do programa e estrutura*. Com as alterações feitas no programa, sua estrutura tende a degradar. Consequentemente, como os programas envelhecem, tornam-se mais difíceis de serem entendidos e alterados. Alguns sistemas foram desenvolvidos sem técnicas modernas de engenharia de software. Eles podem nunca ter sido bem-estruturados e talvez tenham sido otimizados para serem mais eficientes do que inteligíveis. As documentações de sistema podem ter-se perdido ou ser inconsistentes. Os sistemas mais antigos podem não ter sido submetidos a um gerenciamento rigoroso de configuração, então se desperdiça muito tempo para encontrar as versões certas dos componentes do sistema para a mudança.

Figura 9.8 Custo de desenvolvimento e manutenção

Os primeiros três problemas decorrem do fato de que muitas organizações ainda consideram o desenvolvimento e a manutenção como atividades separadas. Manutenção é vista como uma atividade de segunda classe, e não há incentivo para gastar dinheiro, durante o desenvolvimento, para reduzir os custos na alteração do sistema. A única solução a longo prazo para esse problema é aceitar que os sistemas raramente têm um tempo de vida definido, mas continuam em uso, de alguma forma, por um período indeterminado. Como sugeri na introdução, você deve pensar em sistemas evoluindo ao longo de sua vida por um processo contínuo de desenvolvimento.

O quarto item, o problema da estrutura do sistema degradado, é o problema mais fácil de se resolver. Técnicas de reengenharia de software (descritas adiante neste capítulo) podem ser aplicadas para melhorar a estrutura do sistema e sua inteligibilidade. Transformações de arquitetura podem adaptar o sistema para um novo hardware. A refatoração pode melhorar a qualidade do código do sistema e facilitar a mudança.

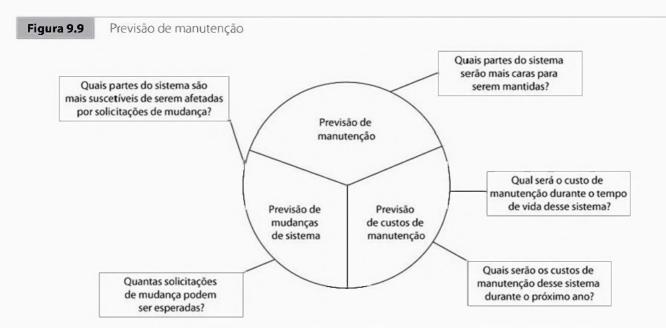
9.3.1 Previsão de manutenção

Gerentes odeiam surpresas, especialmente quando resultam em elevados custos inesperados. Você deve, portanto, tentar prever quais mudanças no sistema podem ser propostas e que partes do sistema são, provavelmente, as mais difíceis de serem mantidas. Você também deve tentar estimar os custos globais de manutenção para um sistema em determinado período de tempo. A Figura 9.9 mostra essas previsões e questões associadas.

Prever o número de solicitações de mudança para um sistema requer uma compreensão do relacionamento entre o sistema e seu ambiente externo. Alguns sistemas possuem um relacionamento muito complexo com seu ambiente externo, e as mudanças nesse ambiente inevitavelmente resultam em alterações. Para avaliar os relacionamentos entre um sistema e seu ambiente, você deve avaliar:

1. *O número e a complexidade das interfaces de sistema.* Quanto maior o número de interfaces e mais complexas elas forem, maior a probabilidade de serem exigidas as alterações de interface quando novos requisitos forem propostos.
2. *O número de requisitos inerentemente voláteis de sistema.* Como discutido no Capítulo 4, os requisitos que refletem as políticas e procedimentos organizacionais são provavelmente mais voláteis do que requisitos baseados em características estáveis de domínio.
3. *Os processos de negócio em que o sistema é usado.* Como processos de negócios evoluem, eles geram solicitações de mudança de sistema. Quanto mais processos de negócios usarem um sistema, maior a demanda por mudanças.

Por muitos anos, os pesquisadores analisaram os relacionamentos entre a complexidade do programa, medida por métricas como a complexidade ciclomática (McCABE, 1976), e sua manutenibilidade (BANKER et al., 1993 ; COLEMAN et al., 1994;. KAFURA e REDDY, 1987; KOZLOV et al., 2008). Não é de se estranhar que esses estudos tenham

Figura 9.9 Previsão de manutenção

revelado que, quanto mais complexo for um sistema ou componente, mais cara será sua manutenção. Medidas de complexidade são particularmente úteis na identificação de componentes de programa suscetíveis a altos custos de manutenção. Kafura e Reddy (1987) analisaram um conjunto de componentes de sistema e descobriram que o esforço de manutenção tende a se centrar em um pequeno número de componentes complexos. Para reduzir os custos de manutenção, portanto, você deve tentar substituir componentes complexos de sistema com alternativas mais simples.

Depois que um sistema foi colocado em serviço, você pode ser capaz de usar dados de processo para ajudar a prever a manutenibilidade. Exemplos de métricas de processo que podem ser usadas para avaliação de manutenibilidade são apresentados a seguir:

1. *Número de solicitações de manutenção corretiva.* Um aumento no número de relatórios de *bugs* e falhas pode indicar que mais erros estão sendo introduzidos no programa do que corrigidos durante o processo de manutenção. Isso pode indicar um declínio na manutenibilidade.
2. *O tempo médio necessário para a análise de impacto.* Isso reflete o número de componentes de programa que são afetados pela solicitação de mudança. Se esse tempo aumenta, isso implica que cada vez mais componentes estão sendo afetados e a manutenibilidade está diminuindo.
3. *O tempo médio gasto para implementar uma solicitação de mudança.* Esse não é o mesmo que o tempo para análise de impacto, embora possam ser correlacionados. Essa é a quantidade de tempo que você precisa para modificar o sistema e sua documentação, depois de ter avaliado quais componentes são afetados. Aumento no tempo necessário para implementar uma mudança pode indicar declínio na manutenibilidade.
4. *Número de solicitações de mudança pendentes.* Ao longo do tempo, um aumento nesse número pode implicar uma diminuição na manutenibilidade.

Você usa informações previstas sobre mudanças de requisitos e previsões sobre a manutenibidade de sistema para prever os custos de manutenção. A maioria dos gerentes combina essas informações com a intuição e a experiência para estimar os custos. O modelo de estimativa de custos COCOMO 2 (BOEHM et al., 2000), discutido no Capítulo 24, sugere que uma estimativa para o esforço de manutenção de software pode ser baseada em um esforço para entender o código existente e os esforços para desenvolver o novo código.

9.3.2 Reengenharia de software

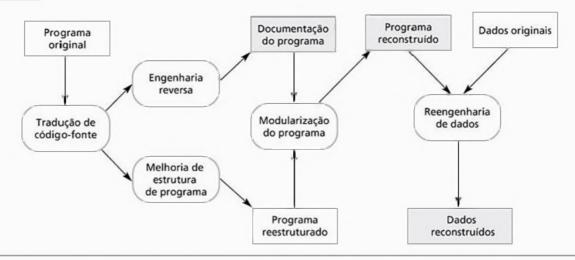
Como discutido na seção anterior, o processo de evolução de sistema envolve a compreensão do programa que tem de ser mudado e, em seguida, a implementação dessas mudanças. No entanto, muitos sistemas, especialmente sistemas legados mais velhos, são difíceis de serem compreendidos e mudados. Os programas podem ter sido otimizados para o desempenho ou uso de espaço à custa de inteligibilidade, ou, ao longo do tempo, a estrutura inicial do programa pode ter sido danificada por uma série de mudanças.

Para fazer com que os sistemas legados de software sejam mais fáceis de serem mantidos, é preciso aplicar reengenharia nesses sistemas visando a melhoria de sua estrutura e inteligibilidade. A reengenharia pode envolver a redocumentação de sistema, a refatoração da arquitetura de sistema, a mudança de linguagem de programação para uma linguagem moderna e modificações e atualizações da estrutura e dos dados de sistema. A funcionalidade de software não é alterada, e você geralmente deve evitar grandes mudanças na arquitetura de sistema.

Existem dois benefícios importantes na reengenharia, em vez de substituição:

1. *Risco reduzido.* Existe um alto risco em desenvolver novamente um software crítico de negócios. Podem ocorrer erros na especificação de sistema ou pode haver problemas de desenvolvimento. Atrasos no início do novo software podem significar a perda do negócio e custos adicionais.
2. *Custo reduzido.* O custo de reengenharia pode ser significativamente menor do que o de desenvolvimento de um novo software. Ulrich (1990) cita um exemplo de um sistema comercial cujos custos de reimplementação foram estimados em 50 milhões de dólares. O sistema foi reconstruído com sucesso por 12 milhões de dólares. Suspeito que, com a tecnologia moderna de software, o custo relativo de reimplementação é provavelmente inferior a esse, mas ainda consideravelmente superior aos custos da reengenharia.

A Figura 9.10 é um modelo geral de processo de reengenharia. A entrada para o processo é um programa legado, e a saída, uma versão melhorada e reestruturada do mesmo programa. As atividades desse processo de reengenharia são:

Figura 9.10 O processo de reengenharia

1. *Tradução de código-fonte.* Usando uma ferramenta de tradução, o programa é convertido a partir de uma linguagem de programação antiga para uma versão mais moderna da mesma linguagem ou em outra diferente.
2. *Engenharia reversa.* O programa é analisado e as informações são extraídas a partir dele. Isso ajuda a documentar sua organização e funcionalidade. Esse processo também é completamente automatizado.
3. *Melhoria de estrutura de programa.* A estrutura de controle do programa é analisada e modificada para que se torne mais fácil de ler e entender. Isso pode ser parcialmente automatizado, mas, normalmente, alguma intervenção manual é exigida.
4. *Modularização de programa.* Partes relacionadas do programa são agrupadas, e onde houver redundância, se apropriado, esta é removida. Em alguns casos, esse estágio pode envolver refatoração de arquitetura (por exemplo, um sistema que usa vários repositórios de dados diferentes pode ser refeito para usar um único repositório). Esse é um processo manual.
5. *Reengenharia de dados.* Os dados processados pelo programa são alterados para refletir as mudanças de programa. Isso pode significar a redefinição dos esquemas de banco de dados e a conversão do banco de dados existente para a nova estrutura. Normalmente devem-se limpar os dados, o que envolve encontrar e corrigir erros, remover registros duplicados etc. Ferramentas são disponíveis para dar suporte à reengenharia de dados.

A reengenharia de programa pode não exigir necessariamente todas as etapas da Figura 9.10. Caso você ainda use o ambiente de desenvolvimento da linguagem de programação, você não precisa da tradução do código-fonte. Se você puder fazer automaticamente a reengenharia, a recuperação de documentação por meio da engenharia reversa pode ser desnecessária. A reengenharia de dados só é necessária se as estruturas de dados de programa mudarem durante a reengenharia de sistema.

Como discutido no Capítulo 19, para fazer o sistema que passou pela reengenharia interoperar com o novo software, você pode precisar desenvolver serviços adaptores. Eles escondem as interfaces originais do sistema de software e apresentam as novas interfaces, bem mais estruturadas e passíveis de serem usadas por outros componentes. Esse processo de empacotamento do sistema legado é uma técnica importante para o desenvolvimento em larga escala de serviços reusáveis.

Os custos da reengenharia, obviamente, dependem da extensão do trabalho. Como mostra a Figura 9.11, existe um espectro de possíveis abordagens para a reengenharia. Os custos aumentam da esquerda para a direita, de modo que a tradução de código-fonte é a opção mais barata. A reengenharia como parte da migração da arquitetura é a mais cara.

O problema com a reengenharia de software é que existem limites práticos para o quanto você pode melhorar um sistema por meio da reengenharia. Não é possível, por exemplo, converter um sistema escrito por meio de uma abordagem funcional para um sistema orientado a objetos. As principais mudanças de arquitetura ou a reorganização radical do sistema de gerenciamento de dados não podem ser feitas automaticamente, pois são muito caras. Embora a reengenharia possa melhorar a manutenibilidade, o sistema reconstruído provavelmente não será tão manutenível como um novo sistema, desenvolvido por meio de métodos modernos de engenharia de software.

Figura 9.11 Abordagens de reengenharia

9.3.3 Manutenção preventiva por refatoração

A refatoração é o processo de fazer melhorias em um programa para diminuir a degradação gradual resultante das mudanças (OPDYKE e JOHNSON, 1990). Isso significa modificar um programa para melhorar sua estrutura, para reduzir sua complexidade ou para torná-lo mais compreensível. No desenvolvimento orientado a objeto, a refatoração muitas vezes é considerada limitada, mas seus princípios podem ser aplicados a qualquer abordagem de desenvolvimento. Quando você refatorar um programa, não deve adicionar funcionalidade, mas concentrar-se na melhoria dele. Portanto, você pode pensar em refatoração como uma 'manutenção preventiva', que reduz os problemas de mudança no futuro.

Embora tanto a reengenharia como a refatoração sejam destinadas a tornar o software mais fácil de entender e mudar, elas não são a mesma coisa. A reengenharia ocorre depois que um sistema foi mantido por algum tempo e com o aumento dos custos de manutenção. Você pode usar ferramentas automatizadas para processar e reestruturar um sistema legado para criar um novo sistema mais manutenível. A refatoração é um processo contínuo de melhoria ao longo do processo de desenvolvimento e evolução, com o intuito de evitar a degradação do código, que aumenta os custos e as dificuldades de manutenção de um sistema.

A refatoração é uma parte inerente dos métodos ágeis, como o Extreme Programming, pois esses métodos são baseados em mudanças. Portanto, a qualidade de programa é suscetível de degradar rapidamente, de modo que os desenvolvedores frequentemente refatoram seus programas para evitar essa degradação. Em métodos ágeis, a ênfase em testes de regressão reduz o risco de introdução de novos erros por meio da refatoração. Quaisquer erros introduzidos deveriam ser detectáveis pelos testes que anteriormente foram bem-sucedidos e deveriam falhar. Entretanto, a refatoração não é dependente de outras 'atividades ágeis' e pode ser usada com qualquer abordagem de desenvolvimento.

Fowler et al. (1999) sugerem que existam situações estereotipadas (que ele chama 'maus cheiros') nas quais o código de um programa pode ser melhorado. Exemplos de maus cheiros que podem ser melhorados por meio de refatoração incluem:

1. *Código duplicado.* O mesmo código ou um muito semelhante pode ser incluído em diferentes lugares de um programa. Este pode ser removido e implementado com um único método ou função que usamos quando necessário.
2. *Métodos longos.* Se um método for muito longo, ele deve ser reprojetado como uma série de métodos mais curtos.
3. *Declarações* switch (case). Geralmente, envolvem a duplicação em situações em que o *switch* depende do tipo de algum valor. As declarações de *switch* podem ser espalhadas em torno de um programa. Em linguagens orientadas a objetos, muitas vezes você pode usar o polimorfismo para conseguir os mesmos resultados.
4. *Aglutinação de dados.* A aglutinação de dados ocorre quando um mesmo grupo de itens de dados (campos em classes, parâmetros em métodos) reincide em vários lugares de um programa. Muitas vezes, eles podem ser substituídos por um objeto que encapsule todos os dados.
5. *Generalidade especulativa.* Isso ocorre quando os desenvolvedores incluem generalidades em um programa, pois estas podem ser necessárias no futuro. Muitas vezes, elas podem ser removidas.

Em seu livro e site, Fowler também sugere algumas transformações primitivas de refatoração para lidar com o mau cheiro, as quais podem ser usadas isoladamente ou em conjunto. Exemplos dessas transformações incluem o método Extract, em que você remove dados duplicados e cria um novo método; a expressão condicional Consolidate, em que você substitui uma sequência de testes por um único teste; e o método Pull up, em que você substitui os métodos similares em subclasses por um único método em uma superclasse. Ambientes de desenvolvimento interativo, como o Eclipse, incluem suporte à refatoração em seus editores e isso facilita encontrar as partes dependentes de um programa que precisam ser alteradas para implementar a refatoração.

A refatoração, quando realizada durante o desenvolvimento de programa, é uma forma eficaz de reduzir os custos de manutenção a longo prazo de um programa. Entretanto, se você assumir um programa para manutenção cuja estrutura foi significativamente degradada, então pode ser praticamente impossível refatorar o código sozinho. É possível que você também tenha de pensar sobre a refatoração de projeto, a qual, provavelmente, é um problema mais caro e difícil. A refatoração de projeto envolve a identificação de padrões de projetos relevantes (discutido no Capítulo 7) e substituir o código existente por um código que implementa esses padrões de projeto (KERIEVSKY, 2004). Por falta de espaço, eu não discuto esse tema aqui.

9.4 Gerenciamento de sistemas legados

Para novos sistemas de software desenvolvidos por meio de processos modernos de engenharia de software, como o desenvolvimento incremental e CBSE, é possível planejar como integrar o desenvolvimento do sistema e sua evolução. Mais e mais empresas estão começando a entender que o processo de desenvolvimento de um sistema é um processo integral de ciclo de vida e que uma separação artificial entre desenvolvimento de software e manutenção de software é inútil. No entanto, ainda existem muitos sistemas legados que são sistemas críticos de negócio. Eles precisam ser ampliados e adaptados às mudanças das práticas de *e-business*.

Geralmente, as organizações têm um portfólio dos sistemas legados que usam, com um orçamento limitado para a manutenção e modernização desses sistemas. Elas precisam decidir como obter o melhor retorno de seus investimentos, o que envolve fazer uma avaliação realista de seus sistemas legados e, em seguida, decidir sobre a estratégia mais adequada para a evolução desses sistemas. Existem quatro opções estratégicas:

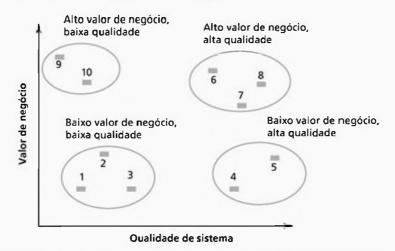
1. *Descartar completamente o sistema*. Essa opção deve ser escolhida quando o sistema não está mais contribuindo efetivamente para os processos dos negócios. Isso geralmente ocorre quando os processos de negócios se alteram desde que o sistema foi instalado e já não são dependentes do sistema legado.
2. *Deixar o sistema inalterado e continuar com a manutenção regular*. Essa opção deve ser escolhida quando o sistema ainda é necessário, mas é bastante estável e os usuários do sistema fazem poucas solicitações de mudança.
3. *Reestruturar o sistema para melhorar sua manutenibilidade*. Essa opção deve ser escolhida quando a qualidade do sistema foi degradada pelas mudanças, e novas mudanças para o novo sistema ainda estão sendo propostas. Esse processo pode incluir o desenvolvimento de novos componentes de interface, para que o sistema original possa trabalhar com outros sistemas mais novos.
4. *Substituir a totalidade ou parte do sistema por um novo sistema*. Essa opção deve ser escolhida quando fatores como hardwares novos significam que o sistema antigo não pode continuar em operação ou quando sistemas de prateleira podem permitir o desenvolvimento do novo sistema a um custo razoável. Em muitos casos, uma estratégia de substituição evolutiva pode ser adotada, na qual, sempre que possível, os componentes principais do sistema são substituídos por sistemas de prateleira com outros componentes reusados.

Naturalmente, essas opções não são exclusivas. Quando um sistema é composto de vários programas, várias opções podem ser aplicadas a cada programa.

Quando você está avaliando um sistema legado, precisa olhar para isso de uma perspectiva de negócio e de uma perspectiva técnica (WARREN, 1998). De uma perspectiva de negócio, você precisa decidir se o negócio realmente necessita do sistema. De uma perspectiva técnica, você precisa avaliar a qualidade do software de aplicação e o apoio de software e hardware do sistema. Em seguida, deve usar uma combinação do valor de negócio e da qualidade de sistema para informar sua decisão sobre o que deve ser feito com o sistema legado.

Por exemplo, suponha que uma organização tenha dez sistemas legados. Você deve avaliar a qualidade e o valor de negócio de cada um e, então, pode criar um gráfico mostrando o valor de negócio relativo e qualidade de sistema, conforme a Figura 9.12.

A partir da Figura 9.12, você pode ver que existem quatro grupos de sistemas:

Figura 9.12 Um exemplo de uma avaliação de sistema legado

1. *Baixa qualidade, baixo valor de negócio.* Manter esses sistemas em funcionamento se tornará caro, e a taxa de retorno para o negócio será bastante reduzida. Esses sistemas devem ser descartados.
2. *Baixa qualidade, alto valor de negócio.* Esses sistemas dão uma contribuição importante para o negócio, portanto, eles não podem ser descartados. Contudo, sua baixa qualidade significa que seu custo de manutenção é alto. Esses sistemas devem ser reestruturados para melhorar a qualidade. Eles podem ser substituídos, caso um sistema de prateleira esteja disponível.
3. *Alta qualidade, baixo valor de negócio.* Esses sistemas não contribuem muito para o negócio, mas seus custos de manutenção podem não ser altos. Não vale a pena substituir esses sistemas; assim, a manutenção normal do sistema pode ser mantida se mudanças de alto custo não forem necessárias e o hardware do sistema permanecer em uso. Se as mudanças necessárias ficarem caras, o software deve ser descartado.
4. *Alta qualidade, alto valor de negócio.* Esses sistemas precisam ser mantidos em operação. No entanto, sua alta qualidade significa que não há necessidade de investimentos na transformação ou substituição do sistema. A manutenção normal do sistema deve ser mantida.

Para avaliar o valor de negócio de um sistema, você precisa identificar os *stakeholders* do sistema, como os usuários finais do sistema e seus gerentes, e fazer uma série de perguntas sobre eles. Existem quatro questões básicas que você precisa discutir:

1. *O uso do sistema.* Se os sistemas são usados apenas ocasionalmente ou por um pequeno número de pessoas, eles podem ter um valor de negócio baixo. Um sistema legado pode ter sido desenvolvido para suprir uma necessidade de negócio que tenha se alterado ou que agora pode ser atendida de formas mais eficazes. No entanto, você precisa ter cuidado acerca de usos ocasionais, mas importantes, do sistema. Por exemplo, em uma universidade, um sistema de registro do aluno só pode ser usado no início de cada ano letivo. No entanto, trata-se de um sistema essencial com um alto valor de negócio.
2. *Os processos de negócios que são apoiados.* Quando um sistema é introduzido, processos de negócios são desenvolvidos para explorar as capacidades dele. Se o sistema for inflexível, mudar os processos pode ser impossível. No entanto, como o ambiente muda, os processos originais de negócio podem tornar-se obsoletos. Portanto, um sistema pode ter um valor de negócio baixo, porque obriga o uso de processos de negócios ineficientes.
3. *Confiança do sistema.* A confiança do sistema não é apenas um problema técnico, mas também um problema de negócio. Se um sistema não for confiável e os problemas afetarem diretamente os clientes do negócio ou significar que as pessoas no negócio são desviadas de outras tarefas para resolverem esses problemas, o sistema terá um valor de negócio baixo.
4. *As saídas do sistema.* A questão-chave aqui é a importância das saídas do sistema para o bom funcionamento do negócio. Se o negócio depender dessas saídas, o sistema terá um alto valor de negócio. Inversamente, se essas saídas puderem ser facilmente geradas de outra forma ou se o sistema produzir saídas que são raramente usadas, então seu valor de negócio poderá ser baixo.

Por exemplo, suponha que uma empresa ofereça um sistema de pedidos de viagem, usado pela equipe responsável pela organização de viagens. Eles podem fazer pedidos com um agente de viagem aprovado. Os bilhetes são entregues e a empresa é cobrada por eles. No entanto, uma avaliação de valor do negócio pode revelar que esse sistema é usado apenas para uma porcentagem bastante pequena dos pedidos colocados; as pessoas que fazem pedidos de viagens acham mais barato e mais conveniente lidar diretamente com os fornecedores de viagens por meio de seus sites. Esse sistema ainda pode ser usado, mas não há motivos reais para que seja mantido. A mesma funcionalidade está disponível em sistemas externos.

Por outro lado, digamos que uma empresa tenha desenvolvido um sistema que mantém o controle de todos os pedidos anteriores de clientes e automaticamente gera lembretes, para que os clientes repitam os pedidos. Esse procedimento resulta em um grande número de pedidos repetidos e mantém os clientes satisfeitos porque sentem que seu fornecedor está ciente de suas necessidades. As saídas de um sistema desse tipo são muito importantes para o negócio e, portanto, esse sistema tem um alto valor de negócio.

Para avaliar um sistema de software a partir de uma perspectiva técnica, é preciso considerar tanto o sistema de aplicação em si, quanto o ambiente no qual ele opera. O ambiente inclui o hardware e todos os softwares de apoio associados (compiladores, ambientes de desenvolvimento etc.) necessários para manter o sistema. O ambiente é importante porque muitas mudanças de sistema resultam de mudanças no ambiente, como atualizações no hardware ou sistema operacional.

Se possível, no processo de avaliação ambiental, você deve fazer as medições de sistema e de seus processos de manutenção. Exemplos de dados que podem ser úteis incluem os custos de manutenção do hardware do sistema e do software de apoio, o número de defeitos de hardware que ocorrem durante um período de tempo e a frequência de *patches* e correções aplicadas ao software de suporte do sistema.

Fatores que você deve considerar na avaliação do ambiente são mostrados na Tabela 9.2. Observe que essas não são todas as características técnicas do ambiente. Você também deve considerar a confiabilidade dos fornecedores de hardware e software de apoio. Se esses fornecedores não estiverem mais no negócio, pode não haver mais suporte para seus sistemas.

Para avaliar a qualidade técnica de um sistema de aplicação, você precisa avaliar uma série de fatores (Tabela 9.3), os quais estão essencialmente relacionados com a confiança do sistema, as dificuldades de sua manutenção e sua documentação. Você também pode coletar dados que o ajudarão a julgar a qualidade do sistema. Dados que podem ser úteis na avaliação de qualidade são:

Tabela 9.2 Fatores usados na avaliação de ambientes

Fator	Questões
Estabilidade do fornecedor	O fornecedor ainda existe? O fornecedor é financeiramente estável e deve continuar existindo? Se o fornecedor não está mais no negócio, existe alguém que mantém os sistemas?
Taxa de falhas	O hardware tem uma grande taxa de falhas reportadas? O software de apoio trava e força o reinício do sistema?
Idade	Quantos anos têm o hardware e o software? Quanto mais velho o hardware e o software de apoio, mais obsoletos serão. Ainda podem funcionar corretamente, mas poderia haver significativos benefícios econômicos e empresariais se migrassem para um sistema mais moderno.
Desempenho	O desempenho do sistema é adequado? Os problemas de desempenho têm um efeito significativo sobre os usuários do sistema?
Requisitos de apoio	Qual apoio local é requisitado pelo hardware e pelo software? Se houver altos custos associados a esse apoio, pode valer a pena considerar a subsituição do sistema.
Custos de manutenção	Quais são os custos de manutenção de hardware e de licenças de software de apoio? Os hardwares mais antigos podem ter custos de manutenção mais elevados do que os sistemas modernos. Os softwares de apoio podem ter altos custos de licenciamento anual.
Interoperabilidade	Existem problemas de interface do sistema com outros sistemas? Compiladores podem, por exemplo, ser usados com as versões atuais do sistema operacional? É necessária a emulação do hardware?

Tabela 9.3 Fatores usados na avaliação de aplicações

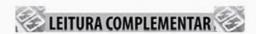
Fatores	Questões
Inteligibilidade	Quão difícil é compreender o código-fonte do sistema atual? Quão complexas são as estruturas de controle usadas? As variáveis têm nomes significativos que refletem sua função?
Documentação	Qual documentação do sistema está disponível? A documentação é completa, consistente e atual?
Dados	Existe um modelo de dados explícito para o sistema? Até que ponto os dados nos arquivos estão duplicados? Os dados usados pelo sistema são atuais e consistentes?
Desempenho	O desempenho da aplicação é adequado? Os problemas de desempenho têm um efeito significativo sobre os usuários do sistema?
Linguagem de programação	Compiladores modernos estão disponíveis para a linguagem de programação usada para desenvolver o sistema? A linguagem de programação ainda é usada para o desenvolvimento do novo sistema?
Gerenciamento de configuração	Todas as versões de todas as partes do sistema são gerenciadas por um sistema de gerenciamento de configuração? Existe uma descrição explícita das versões de componentes usadas no sistema atual?
Dados de teste	Existem dados de teste para o sistema? Existem registros dos testes de regressão feitos quando novos recursos forem adicionados ao sistema?
Habilidades de pessoal	Existem pessoas disponíveis com as habilidades necessárias para manter a aplicação? Existem pessoas disponíveis que tenham experiência no sistema?

1. *O número de solicitações de mudança no sistema.* As alterações no sistema geralmente corrompem a estrutura do sistema e tornam futuras alterações mais difíceis. Quanto maior esse valor acumulado, menor é a qualidade do sistema.
2. *O número de interfaces de usuário.* Esse é um fator importante nos sistemas baseados em formulários, em que cada formulário pode ser considerado uma interface de usuário independente. Quanto mais interfaces, mais prováveis as inconsistências e redundâncias nessas interfaces.
3. *O volume de dados usados pelo sistema.* Quanto maior o volume de dados (número de arquivos, o tamanho do banco de dados etc.), maior a possibilidade de inconsistência nos dados, o que reduz a qualidade do sistema.

Idealmente, a avaliação objetiva deve ser usada para informar as decisões sobre o que fazer com um sistema legado. No entanto, em muitos casos, as decisões não são realmente objetivas, mas baseadas em considerações organizacionais ou políticas. Por exemplo, se duas empresas se fundem, o parceiro politicamente mais poderoso costuma manter seu sistema e se desfazer dos outros. Se a gerência sênior de uma organização decide mudar para uma nova plataforma de hardware, pode ser necessário que as aplicações sejam substituídas. Se não houver orçamento disponível para a transformação do sistema em um determinado ano, então a manutenção do sistema pode continuar, mesmo que isso resulte, a longo prazo, em maiores custos.

PONTOS IMPORTANTES

- O desenvolvimento e a evolução de software podem ser pensados como um processo integrado e interativo, que pode ser representado por um modelo em espiral.
- Para sistemas customizados, os custos de manutenção de software geralmente excedem os custos de desenvolvimento de software.
- O processo de evolução do software é dirigido pelas solicitações de mudança e inclui a análise do impacto da mudança, o planejamento de *release* e implementação da mudança.
- As leis de Lehman, como a noção de que a mudança é contínua, descrevem uma série de considerações provenientes de estudos, de longo prazo, de evolução de sistema.
- Existem três tipos de manutenção de software, ou seja, correção de *bugs*, modificação do software para funcionar em um novo ambiente e implementação de requisitos novos ou alterados.

- A reengenharia de software preocupa-se com a reestruturação e redocumentação do software para torná-lo mais fácil de se entender e mudar.
- Refatoração e pequenas alterações no programa que preservam sua funcionalidade, podem ser pensadas como manutenção preventiva.
- O valor de negócio de um sistema legado e a qualidade do software de aplicação e seu ambiente devem ser avaliados para determinar se o sistema deve ser substituído, transformado ou mantido.

'Software Maintenance and Evolution: A Roadmap'. Além de discutir os desafios da pesquisa, esse artigo apresenta uma breve visão da manutenção e evolução de software pelos principais pesquisadores da área. Os problemas de pesquisa que eles identificaram ainda não foram resolvidos. (RAJLICH, V.; BENNETT, K. H. Proc. 20th Int. Conf. Software Engineering, *IEEE Press*, 2000.) Disponível em: <http://doi.acm.org/10.1145/336512.336534>.

Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices. Esse excelente livro trata de assuntos gerais da manutenção e evolução de software, bem como da migração de sistemas legados. O livro é baseado em um grande estudo de caso da transformação de um sistema COBOL em um sistema cliente-servidor baseado em Java. (SEACORD, R. C.; PLAKOSH, D.; LEWIS, G. A. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley, 2003.)

Working Effectively with Legacy Code. Conselhos práticos e sólidos sobre os problemas e dificuldades de lidar com sistemas legados. (M. FEATHERS, John Wiley & Sons, 2004.)

9.1 Explique por que um sistema de software usado em um ambiente real deve mudar ou tornar-se progressivamente menos útil.

9.2 Explique a base lógica das leis de Lehman. Em que circunstâncias essas leis podem ser quebradas?

9.3 A partir da Figura 9.4, você pode ver que a análise de impacto é um subprocesso importante no processo de evolução de software. Usando um diagrama, sugira quais atividades podem estar envolvidas na análise do impacto de mudanças.

9.4 Como gerente de projeto de software em uma empresa especializada no desenvolvimento de software para a indústria de petróleo *offshore*, você recebeu a tarefa de descobrir os fatores que afetam a manutenibilidade dos sistemas desenvolvidos por sua empresa. Sugira como se pode configurar um programa para analisar o processo de manutenção e descubra as métricas de manutenibilidade adequadas para sua empresa.

9.5 Descreva brevemente os três principais tipos de manutenção de software. Por que às vezes é difícil distingui-los?

9.6 Quais são os principais fatores que afetam os custos de reengenharia de sistema?

9.7 Em que circunstâncias uma organização pode decidir descartar um sistema, mesmo que a avaliação de sistema sugira que ele é de alta qualidade e de alto valor de negócio?

9.8 Quais são as opções estratégicas para a evolução do sistema legado? Quando você substituiria a totalidade ou parte de um sistema, em vez de continuar a manutenção do software?

9.9 Explique por que os problemas com o software de apoio podem significar que uma organização precisa substituir seus sistemas legados.

9.10 Os engenheiros de software têm a responsabilidade profissional de produzir códigos que possam ser mantidos e alterados mesmo que isso não seja explicitamente solicitado por seu empregador?

REFERÊNCIAS

ARTHUR, L. J. *Software Evolution*. Nova York: John Wiley & Sons, 1988.

BANKER, R. D.; DATAR, S. M., KEMERER, C. F.; ZWEIG, D. Software Complexity and Maintenance Costs. *Comm. ACM*, v. 36, n. 11, 1993, p. 81-94.

BOEHM, B. W.; ABTS, C.; BROWN, A. W.; CHULANI, S.; CLARK, B. K.; HOROWITZ, E. et al. *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice Hall, 2000.

COLEMAN, D.; ASH, D.; LOWTHER, B.; OMAN, P. Using Metrics to Evaluate Software System Maintainability'. *IEEE Computer*, v. 27, n. 8, 1994, p. 44-49.

ERLIKH, L. Leveraging legacy system dollars for E-business. *IT Professional*, v. 2, n. 3, mai./jun. 2000, p. 17-23.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. *Refactoring*: Improving the Design of Existing Code. Boston: Addison-Wesley, 1999.

GUIMARAES, T. Managing Application Program Maintenance Expenditures. *Comm. ACM*, v. 26, n. 10, 1983, p. 739-746.

HOPKINS, R.; JENKINS, K. *Eating the IT Elephant*: Moving from Greenfield Development to Brownfield. Boston: IBM Press, 2008.

KAFURA, D.; REDDY, G. R. The use of software complexity metrics in software maintenance. *IEEE Trans. on Software Engineering*, v. SE-13, n. 3, 1987, p. 335-343.

KERIEVSKY, J. *Refactoring to Patterns*. Boston: Addison-Wesley, 2004.

KOZLOV, D.; KOSKINEN, J.; SAKKINEN, M.; MARKKULA, J. Assessing maintainability change over multiple software releases. *J. of Software Maintenance and Evolution*, v. 20, n. 1, 2008, p. 31-58.

KROGSTIE, J.; JAHR, A.; SJOBERG, D. I. K. A longitudinal study of development and maintenance in Norway: Report from the 2003 investigation. *Information and Software Technology*, v. 48, n. 11, 2005, p. 993-1005.

LEHMAN, M. M. Laws of Software Evolution Revisited. *Proc. European Workshop on Software Process Technology* (EWSPT'96). Springer-Verlag, 1996, p. 108-124.

LEHMAN, M. M.; BELADY, L. *Program Evolution*: Processes of Software Change. Londres: Academic Press, 1985.

LEHMAN, M. M.; PERRY, D. E.; RAMIL, J. F. On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution. *Proc. Metrics '98*, Bethesda, Maryland: IEEE Computer Society Press, 1998, p. 84-88.

LEHMAN, M. M.; RAMIL, J. F.; SANDLER, U. An Approach to Modelling Long-term Growth Trends in Software Systems. *Proc. Int. Conf. on Software Maintenance*, Florença, Itália, 2001, p. 219-228.

LIENTZ, B. P.; SWANSON, E. B. *Software Maintenance Management*. Reading, Mass.: Addison-Wesley, 1980.

McCABE, T. J. A complexity measure. *IEEE Trans. on Software Engineering*, v. SE-2, n. 4, 1976, p. 308-320.

NOSEK, J. T.; PALVIA, P. Software maintenance management: changes in the last decade. *Software Maintenance*: Research and Practice, v. 2, n. 3, 1990, p. 157-174.

OPDYKE, W. F.; JOHNSON, R. E. Refactoring: An Aid in Designing Application Frameworks and Evolving Object--Oriented Systems. *1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications* (SOOPPA '90). Poughkeepsie: Nova York, 1990.

POOLE, C.; HUISMAN, J. W. Using Extreme Programming in a Maintenance Environment. *IEEE Software*, v. 18, n. 6, 2001, p. 42-50.

RAJLICH, V. T.; BENNETT, K. H. A Staged Model for the Software Life Cycle. *IEEE Computer*, v. 33, n. 7, 2000, p. 66-71.

SOUSA, M. J. A Survey on the Software Maintenance Process. *14th IEEE International Conference on Software Maintenance* (ICSM '98), Washington, D. C., 1998, p. 265-274.

ULRICH, W. M. The Evolutionary Growth of Software Reengineering and the Decade Ahead. *American Programmer*, v. 3, n. 10, 1990, p. 14-20.

WARREN, I. E. *The Renaissance of Legacy Systems*. Londres: Springer, 1998.

PARTE

1 **2** 3 4

Confiança e proteção

Como os sistemas de software aumentam de tamanho e complexidade, acredito firmemente que nosso maior desafio na engenharia de software é garantir a confiança desses sistemas. Para confiar em um sistema, temos de saber que ele estará disponível quando necessário e executará conforme o esperado. O sistema deve ser protegido para que computadores e dados não sejam ameaçados. Isso significa que as questões relativas à confiança e à proteção do sistema são, muitas vezes, mais importantes que os detalhes de funcionalidade do sistema. Esta parte do livro, portanto, foi concebida para apresentar aos alunos e engenheiros de software tópicos sobre a importância de se ter confiança e proteção.

O primeiro capítulo desta seção, o Capítulo 10, abrange os sistemas sociotécnicos, que, à primeira vista, podem parecer não ter muito a ver com confiança de software. No entanto, muitas falhas de proteção e confiança são provenientes de causas humanas e organizacionais, e não podemos ignorar essas falhas quando consideramos a confiança e a proteção de um sistema. Os engenheiros de software devem estar cientes disso e não devem achar que as melhores técnicas e tecnologias garantem que os sistemas sejam completamente confiáveis e protegidos.

O Capítulo 11 apresenta os conceitos básicos de confiança e proteção e explica os princípios fundamentais de prevenção, detecção e recuperação para construir sistemas confiáveis. O Capítulo 12 suplementa o Capítulo 4, que abrange a engenharia de requisitos, com a discussão de abordagens específicas para derivar e especificar os requisitos de sistema para proteção e confiança. No Capítulo 12, faço uma breve introdução ao uso de especificação formal, e está disponível no site de acompanhamento do livro um capítulo adicional sobre esse tema.

Os capítulos 13 e 14 preocupam-se com as técnicas de engenharia de software para o desenvolvimento de sistemas confiáveis e protegidos. Discuto engenharia de confiança e engenharia de segurança separadamente, mas elas têm muito em comum. Também discuto a importância da arquitetura de software, as diretrizes de projeto atual e as técnicas de programação que nos ajudam a alcançar confiança e proteção. Além disso, explico por que é importante redundância e diversidade para garantir que os sistemas possam lidar com as falhas e ataques externos. Apresento, ainda, o tema extremamente importante da sobrevivência ou resiliência, que permite aos sistemas continuarem a prestar serviços essenciais enquanto sua proteção está sendo ameaçada.

Finalmente, nesta seção, o Capítulo 15 trata da garantia de confiança e proteção. Explico o uso da análise estática e a verificação de modelos para verificação de sistema e detecção de falhas. Essas técnicas têm sido usadas com sucesso na engenharia de sistemas críticos. Eu também cubro abordagens específicas para testar a confiança e a proteção dos sistemas, e explico por que um caso de confiança pode ser necessário para convencer um regulador externo de que um sistema é seguro e protegido.

CAPÍTULO

1 2 3 4 5 6 7 8 9 **10** 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Sistemas sociotécnicos

Objetivos

O objetivo deste capítulo é introduzir o conceito de um sistema sociotécnico — que inclui pessoas, software e hardware — e mostrar o que é necessário para ter uma perspectiva de proteção e confiança do sistema. Com a leitura deste capítulo, você:

- saberá o que se entende por sistema sociotécnico e a diferença entre um sistema técnico, baseado em computador, e um sistema sociotécnico;
- terá sido apresentado ao conceito de propriedade emergente de sistema, tais como confiabilidade, desempenho, segurança e proteção;
- conhecerá as atividades de aquisição, desenvolvimento e operação envolvidas no processo de engenharia de sistemas;
- entenderá por que a confiança e a proteção de software não devem ser tratadas isoladamente e como são afetadas por questões de sistema, tais como erros de operador.

Conteúdo

Em um sistema de computador, o software e o hardware são interdependentes. Sem hardware, um sistema de software é uma abstração, simplesmente uma representação de algum conhecimento e ideias humanas. Sem o software, o hardware é um conjunto de dispositivos eletrônicos inertes. Entretanto, se você os colocar juntos para formar um sistema, criará uma máquina capaz de realizar cálculos complexos e entregar os resultados desses cálculos para seu ambiente.

Isso ilustra uma das características fundamentais de um sistema, que é mais do que a soma de suas partes. Os sistemas têm propriedades que só se tornam aparentes quando seus componentes são integrados e funcionam em conjunto. Portanto, a engenharia de software não é uma atividade isolada, mas uma parte intrínseca de um processo mais geral de processos de engenharia de sistemas. Os sistemas de software não são sistemas isolados, mas componentes essenciais de sistemas mais abrangentes com algum propósito humano, social ou organizacional.

Por exemplo, o software de sistema de controle meteorológico no deserto controla os instrumentos em uma estação meteorológica. Ele se comunica com outros sistemas de software e é uma parte de sistemas de previsão meteorológica nacional e internacional mais amplos. Além do hardware e do software, esses sistemas incluem processos para a previsão do tempo e para as pessoas que operam o sistema e analisam seus resultados. Também incluem as organizações que dependem do sistema para ajudar na previsão do tempo para os indivíduos, governos, indústria etc. Esses sistemas mais amplos são chamados sistemas sociotécnicos. Eles incluem elementos não técnicos, como pessoas, processos, regulamentos etc., bem como componentes técnicos, computadores, software e outros equipamentos.

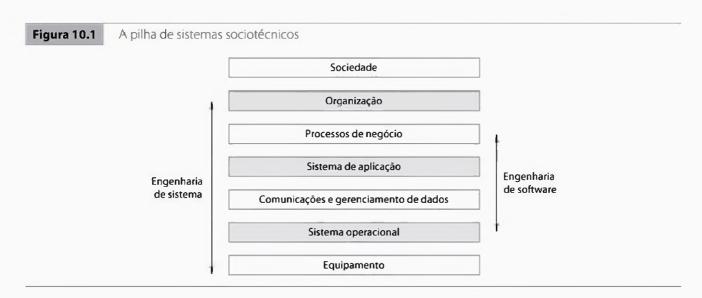
Sistemas sociotécnicos são tão complexos que é praticamente impossível entendê-los como um todo. Em vez disso, você deve percebê-los como camadas, como mostra a Figura 10.1. Essas camadas compõem a pilha de sistemas sociotécnicos:

1. *A camada de equipamentos*. É composta de dispositivos de hardware, alguns dos quais podem ser computadores.
2. *A camada de sistema operacional*. Interage com o hardware e fornece um conjunto de recursos comuns para as camadas superiores de software no sistema.
3. *A camada de comunicações e gerenciamento de dados*. Estende-se até os recursos do sistema operacional e fornece uma interface que permite interação com a mais ampla funcionalidade, como o acesso a sistemas remotos, o acesso ao banco de dados de sistema etc. Algumas vezes, ela é chamada *middleware*, já que está entre a aplicação e o sistema operacional.
4. *A camada de aplicação*. Fornece a funcionalidade específica da aplicação que é requerida. Nela, podem haver muitos programas diferentes de aplicação.
5. *A camada de processos de negócio*. Nesse nível são definidos e aprovados os processos do negócio da organização que usam o sistema de software.
6. *A camada organizacional*. Essa camada inclui processos de alto nível estratégico, bem como regras de negócio, políticas e normas que devem ser seguidas ao se usar o sistema.
7. *A camada social*. Nessa camada estão definidas as leis e os regulamentos da sociedade que governa o funcionamento do sistema.

Em princípio, a maioria das interações ocorre entre as camadas vizinhas, com cada camada superior ocultando os detalhes da inferior. Na prática, nem sempre esse é o caso. Podem haver interações inesperadas entre as camadas, o que resulta em problemas para o sistema como um todo. Por exemplo, digamos que haja uma mudança na lei que regula o acesso às informações pessoais. Essa alteração vem da camada social e leva à necessidade de novos procedimentos organizacionais e de mudanças nos processos dos negócios. No entanto, o sistema de aplicação pode não ser capaz de fornecer o nível de privacidade exigido para que as alterações sejam implementadas na camada de comunicações e gerenciamento de dados.

Ao considerarmos a proteção e confiança do software, é essencial pensarmos de forma holística sobre os sistemas, em vez de apenas considerar o software de forma isolada. As falhas do software *per si* raramente têm consequências sérias, pois o software é intangível e, mesmo quando danificado, é fácil e barato de ser restaurado. Entretanto, quando essas falhas se expandem por meio de outras partes do sistema, elas afetam o ambiente físico e humano do software. Nesse caso, as consequências das falhas são mais significativas. Pode ser necessário que as pessoas façam trabalho extra para conter ou se recuperar da falha, por exemplo, pode haver danos físicos aos equipamentos, os dados podem ser perdidos ou danificados, ou a confidencialidade pode ser quebrada com consequências desconhecidas.

Portanto, ao criarmos um software que precisa ser protegido e confiável, devemos ter uma visão de nível de sistema. Você precisa entender as consequências das falhas do software para outros elementos do sistema. Você também precisa entender como esses outros elementos podem ajudar a se proteger e se recuperar das falhas de software.

Figura 10.1 A pilha de sistemas sociotécnicos

Portanto, o problema real é um sistema, e não uma falha de software, o que significa que você precisa analisar a forma como o software interage com seu ambiente imediato para se assegurar de que:

1. As falhas do software estão, na medida do possível, dentro das camadas do sistema e não afetam gravemente o funcionamento das camadas adjacentes. Em particular, as falhas de software não devem ocasionar a falha do sistema.
2. Você entende como defeitos e falhas nas camadas que não são referentes ao software da pilha de sistemas podem afetar o software. Você também pode considerar como as verificações podem ser incorporadas ao software para ajudar a detectar essas falhas, e como o apoio pode ser fornecido para a recuperação da falha.

Como o software é inerentemente flexível, os problemas inesperados do sistema são frequentemente deixados para que os engenheiros de software resolvam. Digamos que uma instalação de radar tenha sido feita de modo que ocorram fantasmas na imagem. É impraticável mover o radar para um lugar com menos interferências, por isso os engenheiros de sistemas precisam encontrar outra maneira de remover esse efeito fantasma. A solução pode ser melhorar a capacidade de processamento de imagem do software para retirar as imagens fantasmas. Isso pode tornar o software lento e seu desempenho inaceitável. O problema pode ser caracterizado como uma 'falha no software', ao passo que, na realidade, é uma falha no processo de projeto para o sistema como um todo.

É bastante comum esse tipo de situação, na qual engenheiros de software são deixados com o problema de melhorar as capacidades do software sem aumentar os custos de hardware. Muitas vezes, as chamadas falhas de software não são consequência de problemas inerentes ao software, mas resultados da tentativa de mudar o software para acomodar os requisitos de engenharia do sistema alterado. Um bom exemplo disso foi o fracasso do sistema de bagagem do aeroporto de Denver (SWARTZ, 1996), no qual se esperava que o software de controle lidasse com as limitações do equipamento usado.

A engenharia de sistema (STEVENS et al., 1998; THAYER, 2002; THOMÉ, 1993; WHITE et al., 1993) é o processo de projeto de sistemas completos, não apenas o software desses sistemas. O software é um elemento de controle e integração desses sistemas, e frequentemente os custos de engenharia de software são o principal componente do custo global do sistema. Como engenheiro de software, isso ajuda se você tiver maior consciência de como o software interage com outro hardware e sistemas de software, e como estes devem ser usados. Esse conhecimento ajuda a entender os limites do software, a projetar melhor o software e a participar de um grupo de engenharia de sistemas.

10.1 Sistemas complexos

O termo 'sistema' é usado universalmente. Falamos de sistemas de computador, sistemas operacionais, sistemas de pagamento, sistema de ensino, sistema de governo e assim por diante. Essas são aplicações, obviamente muito diferentes da palavra 'sistema', apesar de compartilharem a característica de ser, de alguma forma, mais do que simplesmente a soma de suas partes.

Sistemas abstratos, como os sistemas de governo, estão fora do escopo deste livro. Entretanto, foco a discussão em sistemas que incluam computadores e que tenham algum objetivo específico, como permitir a comunicação, apoiar a navegação ou calcular salários. Uma definição útil desses tipos de sistemas é a seguinte:

> *Um sistema é uma coleção intencional de componentes inter-relacionados, de diferentes tipos, que funcionam em conjunto para atingir um objetivo.*

Essa definição geral abrange uma vasta gama de sistemas. Por exemplo, um sistema simples, como o apontador a *laser*, pode incluir alguns componentes de hardware e pouco além de uma pequena quantidade de software de controle. Em contraste, um sistema de controle aéreo inclui milhares de componentes de hardware e software, além de usuários humanos que tomam decisões com base em informações do sistema computacional.

Uma característica de todos os sistemas complexos é que as propriedades e o comportamento dos componentes do sistema estão inextricavelmente interligados. O bom funcionamento de cada componente do sistema depende do funcionamento de outros componentes. Assim, o software só pode funcionar se o processador estiver em operação. E o processador só pode executar cálculos se o sistema de software que define esses cálculos for instalado com sucesso.

Os sistemas complexos geralmente são hierárquicos e incluem outros sistemas. Por exemplo, um sistema de comando e controle da polícia pode incluir um sistema de informação geográfica para fornecer detalhes do local dos incidentes. Nesses sistemas estão incluídos os chamados 'subsistemas'. Eles podem operar como sistemas independentes, por direito próprio. Por exemplo, o mesmo sistema de informação geográfica pode ser usado em logística de transporte e de comando e controle de emergência.

Sistemas que incluem software encaixam-se em duas categorias:

1. *Sistemas técnicos baseados em computador*. São sistemas que incluem componentes de hardware e de software, mas não os procedimentos e processos. Exemplos de sistemas técnicos incluem televisores, telefones celulares e outros equipamentos com software embutido. A maioria dos softwares para computadores pessoais, jogos de computador etc., também se enquadra nessa categoria. Indivíduos e organizações usam sistemas técnicos para uma finalidade específica, mas o conhecimento desse objetivo não é parte do sistema. Por exemplo, o processador de texto que eu estou usando não está consciente de que ele está sendo usado para escrever um livro.
2. *Sistemas sociotécnicos*. Incluem um ou mais sistemas técnicos, mas, principalmente, também pessoas que entendem o propósito do software dentro do próprio sistema. Os sistemas sociotécnicos definiram que os processos operacionais e as pessoas (os operadores) são partes inerentes do sistema. Eles são regulados por políticas e regras organizacionais e podem ser afetados por restrições externas, como leis e políticas nacionais de regulação. Por exemplo, este livro foi criado por meio de um sistema de publicação sociotécnico que inclui vários processos e sistemas técnicos.

Sistemas sociotécnicos são sistemas corporativos destinados a contribuir para o cumprimento de uma meta de negócios. Esta pode ser aumentar as vendas, reduzir os materiais usados na manufatura, arrecadar impostos, manter um espaço aéreo seguro etc. Como os sistemas sociotécnicos estão embutidos em um ambiente organizacional, a aquisição, o desenvolvimento e o uso desses sistemas são influenciados pelas políticas de procedimentos organizacionais, bem como por sua cultura de trabalho. Os usuários do sistema são pessoas influenciadas pela forma como a organização é gerida e suas interações com outras pessoas dentro e fora da organização.

Quando você está tentando desenvolver sistemas sociotécnicos, precisa entender o ambiente organizacional em que eles serão usados. Se você não fizer isso, os sistemas podem não atender às necessidades de negócios, e os usuários e seus gerentes podem rejeitar o sistema. Os fatores organizacionais do ambiente do sistema que podem afetar os requisitos, projeto e operação de um sistema sociotécnico incluem:

1. *Mudanças de processos*. O sistema pode exigir mudanças nos processos de trabalho do ambiente. Se assim for, um treinamento será certamente necessário. Se as alterações forem significativas ou envolverem demissões, existe o perigo de os usuários resistirem à introdução do sistema.
2. *Mudanças de trabalho*. Novos sistemas podem desqualificar os usuários em um ambiente ou fazê-los mudar a forma como trabalham. Se assim for, os usuários podem resistir ativamente à introdução do sistema na organização. Geralmente, os projetos que envolvem gerentes que têm de mudar a sua maneira de trabalhar para atender a um novo sistema, geralmente ficam aborrecidos. Os gerentes podem sentir que o sistema está reduzindo seu *status* na organização.
3. *Mudanças organizacionais*. O sistema pode alterar a estrutura do poder político em uma organização. Por exemplo, se uma organização for dependente de um sistema complexo, aqueles que controlam o acesso a esse sistema terão grande poder político.

Os sistemas sociotécnicos têm três características particularmente importantes quando se consideram proteção e confiança:

1. Eles têm propriedades emergentes que são propriedades do sistema como um todo, e não associadas apenas a partes individuais do sistema. As propriedades emergentes dependem tanto dos componentes do sistema quanto dos relacionamentos entre eles. Dada essa complexidade, as propriedades emergentes só podem ser avaliadas uma vez que o sistema tenha sido montado. A proteção e a confiança são propriedades de sistema emergente.
2. Frequentemente, eles são não determinísticos. Isso significa que, quando apresentados a uma entrada específica, eles nem sempre produzem a mesma saída. O comportamento do sistema depende dos operadores humanos, e as pessoas nem sempre reagem da mesma maneira. Além disso, o uso do sistema pode criar novos relacionamentos entre os componentes de sistema e, consequentemente, alterar seu comportamento emergente. Os defeitos e falhas de sistema podem ser transitórios, e as pessoas podem discordar sobre a ocorrência de uma falha.
3. Embora o sistema apoie os objetivos organizacionais, estes não dependem apenas do próprio sistema. Eles também dependem da estabilidade desses objetivos, dos relacionamentos e conflitos entre os objetivos organizacionais e da interpretação desses objetivos pelas pessoas na organização. Um novo gerenciamento pode reinterpretar os objetivos organizacionais que o sistema suporta, de modo que um sistema 'bem-sucedido' possa ser visto como um 'fracasso'.

Frequentemente, as considerações sociotécnicas são cruciais para determinar se um sistema cumpriu com êxito seus objetivos. Infelizmente, para os engenheiros com pouca experiência em estudos sociais ou culturais, considerá-las é muito difícil.

Várias metodologias têm sido desenvolvidas para ajudar a entender os efeitos dos sistemas nas organizações, como a sociotécnica de Mumford (1989) e a Soft Systems Methodology de Checkland (1981; CHECKLAND e SCHOLES, 1990). Houve também estudos sociológicos sobre os efeitos dos sistemas baseados em computadores no trabalho (ACKROYD et al., 1992; ANDERSON et al., 1989; SUCHMAN, 1987).

10.1.1 Propriedades emergentes de sistema

Os complexos relacionamentos entre os componentes de um sistema significam que um sistema é mais do que simplesmente a soma de suas partes. Ele tem propriedades próprias, do sistema como um todo. Essas 'propriedades emergentes' (CHECKLAND, 1981) não podem ser atribuídas a qualquer parte específica do sistema. Elas só surgem quando os componentes do sistema são integrados. Algumas dessas propriedades, como o peso, podem ser obtidas diretamente das propriedades comparáveis dos subsistemas. Entretanto, costumam resultar dos complexos inter-relacionamentos dos subsistemas. As propriedades de um sistema não podem ser calculadas a partir das propriedades de seus componentes individuais. Exemplos de algumas propriedades emergentes são mostrados na Tabela 10.1.

Existem dois tipos de propriedades emergentes:

1. Propriedades emergentes funcionais, quando a finalidade do sistema só surge após seus componentes serem integrados. Por exemplo, uma bicicleta, uma vez que montada a partir de seus componentes, tem a propriedade funcional de ser um meio de transporte.
2. Propriedades emergentes não funcionais, que se relacionam com o comportamento do sistema em seu ambiente operacional. A confiabilidade, o desempenho, a segurança e a proteção são exemplos de propriedades emergentes. Esses são fatores críticos para sistemas baseados em computadores. Geralmente, a falha em alcançar um nível mínimo definido nessas propriedades faz com que o sistema se torne inútil. Alguns usuários podem não precisar de alguma das funções de sistema; assim, o sistema pode ser aceitável sem essas funções. No entanto, um sistema não confiável ou muito lento é suscetível de ser rejeitado por todos os usuários.

Propriedades emergentes de confiança, como a confiabilidade, dependem tanto das propriedades dos componentes individuais quanto de suas interações. Os componentes de um sistema são interdependentes. As falhas em um componente podem ser propagadas por meio do sistema e vir a afetar o funcionamento de outros componentes. No entanto, muitas vezes é difícil prever como essas falhas afetarão outros componentes. Portanto, a partir de dados sobre a confiabilidade dos componentes de sistema, é praticamente impossível estimar a confiabilidade do sistema global.

Tabela 10.1 Exemplos de propriedades emergentes

Propriedade	Descrição
Volume	O volume de um sistema (o espaço total ocupado) varia conforme os conjuntos de componentes estão dispostos e conectados.
Confiabilidade	A confiabilidade de sistema depende da confiabilidade de componentes, mas interações inesperadas podem causar novos tipos de falhas e, portanto, afetar a confiabilidade do sistema.
Proteção	A proteção do sistema (sua capacidade de resistir ao ataque) é uma propriedade complexa que não pode ser facilmente mensurada. Os ataques podem ser criados de forma imprevista pelos projetistas de sistemas e, assim, derrotar as proteções internas.
Reparabilidade	Essa propriedade reflete quão fácil é corrigir um problema com o sistema uma vez que este tenha sido descoberto. Depende da capacidade de diagnosticar o problema e do acesso a componentes que estejam com defeito, bem como de se modificar ou substituir tais componentes.
Usabilidade	Essa propriedade reflete quão fácil é usar o sistema. Depende dos componentes técnicos de sistema, seus operadores e seu ambiente operacional.

Em um sistema sociotécnico, você precisa considerar a confiabilidade a partir de três perspectivas:

1. *Confiabilidade de hardware*. Qual é a probabilidade de os componentes de hardware falharem, e quanto tempo leva para reparar um componente que falhou?
2. *Confiabilidade de software*. Qual é a probabilidade de um componente de software produzir uma saída incorreta? Falhas de software são distintas das falhas de hardware, em que o software não se desgasta. Falhas são geralmente transitórias. O sistema continua a funcionar após um resultado incorreto.
3. *Confiabilidade de operador*. Qual é a probabilidade do operador de um sistema cometer um erro e fornecer uma entrada incorreta? Qual é a probabilidade de o software não detectar esse erro e o propagar?

A confiabilidade de hardware, de software e de operador não são independentes. A Figura 10.2 mostra como as falhas em um nível podem ser propagadas para os demais níveis do sistema. Falhas no hardware podem gerar sinais espúrios, fora da faixa de entradas esperada pelo software. Assim, o software pode comportar-se de forma imprevisível e produzir saídas inesperadas, as quais podem confundir e, consequentemente, estressar o operador de sistema.

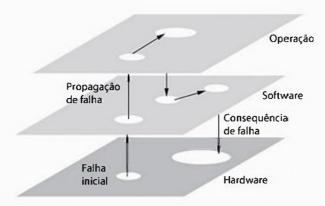
Erro de operador é mais provável quando o operador está se sentindo estressado. Assim, uma falha de hardware pode significar que o operador de sistema comete erros que, por sua vez, podem gerar problemas de software ou processamento adicional. Isso pode sobrecarregar o hardware, causando mais falhas, e assim por diante. Assim, a falha inicial, que poderia ser recuperável, pode rapidamente se transformar em um problema grave, que pode resultar em um desligamento completo do sistema.

A confiabilidade de um sistema depende do contexto em que ele é usado. No entanto, o ambiente de sistema não pode ser completamente especificado, e os projetistas não podem colocar restrições nesse ambiente para sistemas operacionais. Diferentes sistemas que operam em um ambiente podem reagir a problemas de formas imprevisíveis, afetando a confiabilidade de todos esses sistemas.

Por exemplo, digamos que um sistema seja projetado para operar sob temperatura ambiente. Para permitir variações e condições excepcionais, os componentes eletrônicos de um sistema são projetados para operar em um determinado intervalo de temperaturas, digamos, de 0 a 45 graus. Fora dessa faixa de temperatura, os componentes se comportarão de forma imprevisível. Agora, vamos supor que esse sistema seja instalado próximo a um aparelho de ar condicionado. Se esse ar condicionado falhar e ventilar gás quente sobre o sistema eletrônico, o sistema poderá superaquecer. Logo, os componentes e todo o sistema poderão falhar.

Se esse sistema tivesse sido instalado em outro local daquele ambiente, esse problema não teria ocorrido. Quando o ar condicionado funcionou corretamente, não houve problemas. No entanto, devido à proximidade física dessas máquinas, havia um relacionamento inesperado entre elas que levou à falha do sistema.

Assim como a confiabilidade, as propriedades emergentes, como desempenho ou usabilidade, são difíceis de serem avaliadas, mas podem ser medidas uma vez que o sistema esteja em operação. No entanto, propriedades como segurança e proteção não são mensuráveis. Nesse caso, você não está apenas preocupado com os atributos que se relacionam com o comportamento do sistema, mas também com comportamentos indesejáveis ou inaceitáveis. Um sistema protegido é aquele que não permite o acesso não autorizado a seus dados. No entanto, é claramente impossível prever todos os possíveis modos de acesso e proibi-los explicitamente. Portanto, só poderá ser possível avaliar essas propriedades 'não deve' por *default*. Ou seja, você só sabe que um sistema não é protegido quando alguém consegue penetrar no sistema.

Figura 10.2 Propagação da falha

10.1.2 Não determinismo

Um sistema determinístico é aquele que é totalmente previsível. Se ignorarmos problemas de *timing*, os sistemas de software que rodam em hardwares totalmente confiáveis quando apresentados a uma sequência de entradas produzirão sempre a mesma sequência de saídas. Claro, não existe um hardware completamente confiável, mas ele é geralmente confiável o suficiente para pensarmos em sistemas de hardware como determinísticos.

Pessoas, por outro lado, são não determinísticas. Quando apresentadas à mesma entrada (por exemplo, um pedido para concluir uma tarefa), suas respostas dependerão de seu estado emocional e físico, da pessoa que fez o pedido, das outras pessoas no ambiente, de tudo o que eles estão fazendo etc. Em alguns casos, as mesmas pessoas ficarão felizes em realizar o trabalho e, em outros, se recusarão.

Sistemas sociotécnicos são parcialmente não determinísticos, porque incluem pessoas e parcialmente porque as alterações de hardware, software e dados nesses sistemas são muito frequentes. As interações entre essas mudanças são complexas e, por isso, o comportamento do sistema é imprevisível. Esse não é um problema em si, mas, a partir de uma perspectiva de confiança, pode tornar difícil decidir quando ocorre uma falha de sistema e estimar a frequência dessas falhas.

Por exemplo, digamos que um sistema seja apresentado a um conjunto de 20 entradas de testes. Ele processa essas entradas e os resultados são registrados. Algum tempo depois, as mesmas 20 entradas de teste são processadas, e os resultados, comparados com os resultados anteriores armazenados. Cinco delas são diferentes. Isso significa que houve cinco falhas? Ou essas diferenças são simplesmente resultado de variações razoáveis no comportamento do sistema? A única maneira de descobrir é verificando os resultados de forma mais aprofundada e fazendo julgamentos sobre a forma como o sistema tem tratado cada entrada.

10.1.3 Os critérios de sucesso

Geralmente, os sistemas sociotécnicos complexos são desenvolvidos para combater o que, às vezes, são chamados 'problemas graves' (RITTEL e WEBBER, 1973). Um problema grave é um problema que é tão complexo e envolve tantas entidades relacionadas que não existe especificação definitiva dele. Diferentes *stakeholders* percebem o problema de diferentes maneiras e ninguém tem um completo entendimento do problema como um todo. A verdadeira natureza do problema surgirá apenas quando uma solução for desenvolvida. Um exemplo extremo de um problema complexo é o planejamento para terremotos. Ninguém pode prever com precisão onde será o epicentro de um terremoto, a que horas ocorrerá ou o efeito que terá sobre o ambiente. É impossível especificar detalhadamente como lidar com um grande terremoto.

Isso torna difícil definir os critérios de sucesso para um sistema. Como você decide se um novo sistema contribui, como previsto, para os objetivos do negócio da empresa que paga pelo sistema? Geralmente, a análise do sucesso não é feita por meio da análise das razões originais para a aquisição e desenvolvimento do sistema. Pelo contrário, baseia-se na efetividade do sistema no momento em que ele é implantado. Como o ambiente de negócio pode mudar muito rapidamente durante o desenvolvimento do sistema, os objetivos do negócio podem mudar significativamente.

A situação é ainda mais complexa quando existem vários objetivos conflitantes, interpretados de maneiras diversas por diferentes *stakeholders*. Por exemplo, o sistema no qual o MHC-PMS (discutido no Capítulo 1) é baseado foi projetado para suportar dois objetivos distintos de negócios:

1. Melhorar a qualidade do atendimento para pessoas que sofrem de doenças mentais.
2. Aumentar a renda, fornecendo relatórios detalhados dos cuidados prestados e os custos dessa assistência.

Infelizmente, tais objetivos se revelaram conflitantes, pois a informação necessária para satisfazer o objetivo faz que médicos e enfermeiros tenham de fornecer informações adicionais, além dos registros de saúde normalmente mantidos. Isso reduziu a qualidade do atendimento para os pacientes, além de significar que a equipe de clínicos teve menos tempo para falar com os pacientes. Do ponto de vista de um médico, esse sistema não foi uma melhoria do sistema manual anterior, mas da perspectiva de um gerente foi, sim, uma melhoria.

Algumas vezes, a natureza dos atributos de proteção e confiança torna ainda mais difícil decidir se o sistema foi bem-sucedido. A intenção de um novo sistema pode ser melhorar a proteção por meio da substituição de um sistema existente por um ambiente de dados mais seguros. Digamos que, após a instalação, o sistema é atacado, uma quebra de segurança ocorre, e alguns dados ficam corrompidos. Isso significa que o sistema é um fracasso?

Nós não podemos dizer, porque não sabemos a extensão das perdas que teriam ocorrido com o antigo sistema dados os mesmos ataques.

10.2 Engenharia de sistemas

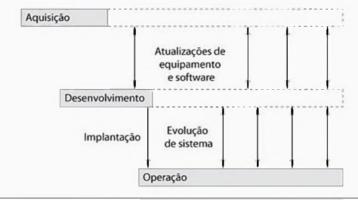
A engenharia de sistemas engloba todas as atividades envolvidas na aquisição, especificação, projeto, implementação, validação, implantação, operação e manutenção dos sistemas sociotécnicos. Os engenheiros de sistemas não estão preocupados apenas com o software, mas também com o hardware e as interações do sistema com os usuários e com seu ambiente. Eles devem pensar sobre os serviços que o sistema oferece, as restrições sob as quais o sistema deve ser construído e operado e as maneiras pelas quais o sistema é usado para cumprir seu propósito ou finalidade.

Durante o período de vida de grandes e complexos sistemas sociotécnicos existem três estágios sobrepostos (Figura 10.3):

1. *Obtenção ou aquisição.* Durante esse estágio, o objetivo de um sistema é decidido, os requisitos do sistema de alto nível são estabelecidos, são tomadas as decisões sobre como a funcionalidade será distribuída entre software, hardware e pessoas, e são comprados os componentes do sistema.
2. *Desenvolvimento.* Durante esse estágio, o sistema é desenvolvido. Os processos de desenvolvimento incluem todas as atividades envolvidas no desenvolvimento do sistema, como definição de requisitos, projeto de sistemas, engenharia de hardware e software, integração de sistemas e testes. Os processos operacionais são definidos e os cursos de treinamento para usuários do sistema são projetados.
3. *Operação.* Nesse estágio, o sistema é implantado, os usuários são treinados e o sistema é colocado em uso. Geralmente, os processos operacionais previstos precisam mudar para refletir o ambiente de trabalho real em que o sistema é usado. Ao longo do tempo, o sistema evolui à medida que são identificados novos requisitos. Eventualmente, o sistema perde valor, é desqualificado e substituído.

Esses estágios não são independentes. Quando o sistema está operando, novos softwares e equipamentos podem precisar ser adquiridos para substituir os componentes do sistema obsoleto, a fim de proporcionar uma nova funcionalidade ou atender à demanda crescente. Da mesma forma, pedidos de alteração durante a operação demandam o desenvolvimento adicional de sistema.

A proteção e a confiança globais de um sistema são influenciadas por atividades em todos esses estágios. As opções de projeto podem ser restringidas por decisões de aquisição no âmbito do sistema, bem como no hardware e software do sistema. Pode ser impossível implementar alguns tipos de salvaguardas do sistema. Estas podem apresentar vulnerabilidades que podem gerar futuras falhas no sistema. Erros humanos ocorridos durante os estágios de especificação, projeto e desenvolvimento podem significar que defeitos foram introduzidos no sistema. Testes inadequados podem significar que os defeitos não foram descobertos antes de o sistema ser implantado. Durante a operação, erros na configuração do sistema para a implantação podem ocasionar novas vul-

Figura 10.3 Estágios da engenharia de sistemas

nerabilidades. Os operadores de sistema podem cometer erros durante seu uso. Quando são feitas alterações no sistema, as considerações feitas durante a aquisição original podem ser esquecidas e, novamente, vulnerabilidades podem ser introduzidas no sistema.

Uma diferença importante entre sistemas e engenharia de software é o envolvimento de uma gama de disciplinas profissionais em toda a vida útil do sistema. Por exemplo, as disciplinas técnicas que podem estar envolvidas na aquisição e desenvolvimento de um novo sistema de gerenciamento de tráfego aéreo são mostradas na Figura 10.4. Arquitetos e engenheiros civis estão envolvidos, pois, geralmente, novos sistemas de gerenciamento do tráfego aéreo precisam ser instalados em um prédio novo. Os engenheiros elétricos e mecânicos são envolvidos para especificar e manter a energia e o ar condicionado. Os engenheiros eletrônicos preocupam-se com os computadores, radares e outros equipamentos. Os ergonomistas projetam as estações de trabalho do controlador e engenheiros de software e projetistas de interface com o usuário são responsáveis pelo software do sistema.

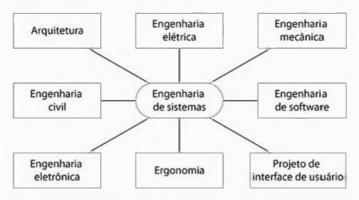
O envolvimento de uma gama de disciplinas profissionais é essencial, pois existem muitos aspectos diferentes dos sistemas sociotécnicos complexos. No entanto, as diferenças entre as disciplinas podem introduzir vulnerabilidades em sistemas e, assim, comprometer a proteção e a confiança do sistema a ser desenvolvido:

1. Diferentes disciplinas usam as mesmas palavras para significar coisas diferentes. Desentendimentos são comuns nos debates entre engenheiros de diferentes origens. Se estes não são descobertos e resolvidos durante o desenvolvimento do sistema, podem levar a erros nos sistemas entregues. Por exemplo, um engenheiro eletrônico pode saber um pouco sobre programação C#, mas pode não compreender que um método em Java é comparável a uma função em C.
2. Cada disciplina faz suposições sobre o que pode ou não pode ser feito por outras disciplinas. Muitas vezes, ainda, são baseadas em uma compreensão inadequada do que é realmente possível. Por exemplo, um projetista de interface de usuário pode propor uma interface gráfica para um sistema embutido que requer grande quantidade de processamento e, assim, sobrecarregar o processador do sistema.
3. As disciplinas tentam proteger suas fronteiras profissionais e podem argumentar contra certas decisões de projeto, pois essas decisões exigem sua especialização profissional. Portanto, um engenheiro de software pode argumentar a favor de um sistema que trava as portas do edifício, embora um sistema mecânico baseado em chaves possa ser mais confiável.

10.3 Aquisição de sistemas

A fase inicial da engenharia de sistemas é a de obtenção do sistema (às vezes chamada aquisição do sistema). Nessa fase, as decisões são tomadas no âmbito de um sistema que está para ser comprado, de orçamentos e cronograma de sistema e dos requisitos de alto nível de sistema. Usando essa informação, são tomadas novas decisões sobre a obtenção de um sistema, o tipo de sistema requerido e o fornecedor ou fornecedores de sistema. Os direcionadores para essas decisões são:

1. *O estado de outros sistemas organizacionais.* Se a organização tem um conjunto de sistemas que não se comunicam facilmente ou que são caros para serem mantidos, a aquisição de um sistema substituto pode levar a significativos benefícios comerciais.

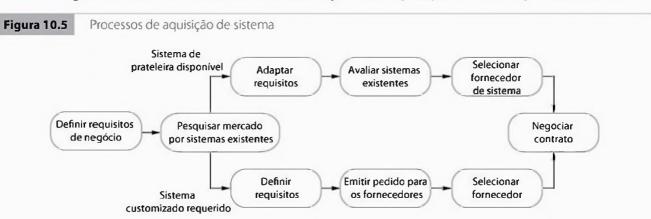
Figura 10.4 Disciplinas profissionais envolvidas na engenharia de sistemas

2. *A necessidade de cumprir com as regulamentações externas.* Os negócios são regulados e precisam demonstrar conformidade com os regulamentos definidos externamente (por exemplo, os regulamentos Sarbanes-Oxley de contabilidade nos Estados Unidos). Isso pode exigir a substituição de sistemas não conformes ou novos sistemas especificamente para monitorar a conformidade com as regulamentações.
3. *Concorrência externa.* Se uma empresa precisa competir de forma mais eficaz ou manter uma posição competitiva, pode ser aconselhável o investimento em novos sistemas que melhorem a eficiência dos processos de negócios. Para os sistemas militares, a necessidade de melhorar a capacidade em face das novas ameaças é uma razão importante para a aquisição de novos sistemas.
4. *Reorganização de negócio.* Negócios e outras organizações frequentemente se reestruturam com o intuito de melhorar a eficiência do serviço e/ou atendimento ao cliente. As reorganizações geram mudanças nos processos de negócios que necessitam de suporte dos novos sistemas.
5. *Orçamento disponível.* O orçamento disponível é um fator óbvio para determinar o escopo dos novos sistemas que podem ser adquiridos.

Além disso, novos sistemas de governo são muitas vezes adquiridos para refletirem mudanças políticas e políticas públicas. Por exemplo, nos Estados Unidos, políticos de grandes potências mundiais podem decidir comprar novos sistemas de vigilância, que, segundo eles, ajudarão na luta contra o terrorismo. A compra desses sistemas mostra que os eleitores estão pressionando. No entanto, tais sistemas são muitas vezes adquiridos sem uma análise da relação custo-benefício, na qual se comparam os benefícios resultantes das diferentes opções de gastos.

Sistemas complexos de grande porte geralmente consistem em uma mistura de componentes de prateleira e os especialmente construídos. Uma razão pela qual cada vez mais softwares são incluídos nos sistemas é que eles permitem um maior uso de componentes de hardware existentes, com o software agindo como 'cola' para fazer esses componentes do hardware trabalharem juntos de forma eficaz. A necessidade de desenvolver essa 'cola' é uma razão pela qual, às vezes, a economia resultante do uso de componentes de prateleira não é tão grande quanto o previsto.

A Figura 10.5 mostra um modelo simplificado do processo de aquisição para os dois tipos, componentes COTS e de sistema, que precisam ser especialmente projetados e desenvolvidos. Pontos importantes sobre o processo mostrado nesse diagrama são:

1. Componentes de prateleira geralmente não correspondem exatamente aos requisitos, a menos que os requisitos tenham sido escritos com esses componentes em mente. Portanto, a escolha de um sistema significa que você precisa encontrar a correspondência mais próxima entre os requisitos de sistema e os recursos oferecidos pelos sistemas de prateleira. Você pode, então, ter de modificar os requisitos, o que pode ter repercussão sobre outros subsistemas.
2. Quando um sistema deve ser especialmente construído, a especificação de requisitos é parte integrante do contrato para o sistema que está sendo adquirido. É, portanto, um documento legal, assim como um documento técnico.
3. Após um contratante ter sido selecionado para construir um sistema, existe um período de negociações, durante o qual você pode precisar negociar novas alterações nos requisitos e discutir questões como o custo das mudanças para o sistema. Da mesma forma, uma vez que um sistema COTS tenha sido selecionado, você pode negociar com o fornecedor sobre os custos, condições de licença, as possíveis mudanças no sistema etc.

Figura 10.5 Processos de aquisição de sistema

Em sistemas sociotécnicos, o software e o hardware normalmente são desenvolvidos por uma organização diferente (o fornecedor) da que está adquirindo todo o sistema sociotécnico. A razão para isso é que o negócio do cliente raramente é o desenvolvimento de software, de modo que seus funcionários não têm as habilidades necessárias para desenvolver os próprios sistemas. Na verdade, poucas empresas têm a capacidade de projetar, fabricar e testar todos os componentes de um sistema sociotécnico complexo de grande porte.

Consequentemente, o fornecedor de sistema, que geralmente é chamado contratante principal, contrata o desenvolvimento de diferentes subsistemas para vários subcontratantes. Para sistemas de grande porte, como sistemas de controle de tráfego aéreo, um grupo de fornecedores pode formar um consórcio para licitação do contrato. O consórcio deve incluir todos os recursos necessários para esse tipo de sistema, abrangendo fornecedores de hardware, desenvolvedores de software, fornecedores de periféricos e fornecedores de equipamentos especializados, como sistemas de radar.

O agente negocia com o contratante, e não com os subcontratantes, para que haja uma única interface agente/fornecedor. Os subcontratantes projetam e constroem as partes do sistema para uma especificação, produzida pelo contratante principal. Depois dessa fase, o contratante principal integra os diferentes componentes e os entrega ao cliente. Dependendo do contrato, o agente pode permitir ao contratante principal a livre escolha dos subcontratantes ou pode exigir que ele selecione a partir de uma lista aprovada.

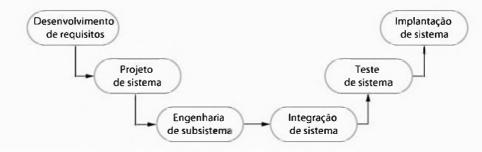
As decisões e escolhas feitas durante a aquisição de sistema têm um efeito profundo sobre sua proteção e confiança. Por exemplo, se for tomada a decisão de adquirir um sistema de prateleira, a organização precisará aceitar que esses sistemas têm influência limitada sobre os requisitos de proteção e confiança do sistema. Eles dependem em grande medida das decisões tomadas pelos vendedores de sistemas. Além disso, os sistemas de prateleira podem ter fraquezas de proteção e confiança já conhecidos ou exigir configurações complexas. Os erros de configuração, em que os pontos de entrada do sistema não são devidamente protegidos, são uma grande fonte de problemas de proteção.

Contudo, a decisão de adquirir um sistema customizado significa que um grande esforço deve ser dedicado à compreensão e definição dos requisitos de proteção e confiança. Se uma empresa tem pouca experiência nessa área, essa decisão será muito difícil. Se o nível requerido de confiança, bem como de desempenho aceitável de sistema, devem ser alcançados, então o tempo de desenvolvimento talvez deva ser estendido, e o orçamento, aumentado.

10.4 Desenvolvimento de sistemas

Os objetivos do processo de desenvolvimento do sistema são desenvolver ou adquirir todos os componentes de um sistema e, em seguida, integrar esses componentes para criar um sistema final. Os requisitos são a ponte entre os processos de aquisição e de desenvolvimento. Durante a aquisição, requisitos de negócio e requisitos de sistema funcionais e não funcionais de alto nível são definidos. Você pode pensar nisso como o início do desenvolvimento, daí a sobreposição de processos mostrados na Figura 10.3. Assim que os contratos sobre os componentes do sistema forem acordados, inicia-se a engenharia de requisitos mais detalhada.

A Figura 10.6 é um modelo do processo de desenvolvimento de sistemas. Esse processo de engenharia de sistemas foi uma importante influência sobre o modelo 'cascata' do processo de software discutido no Capítulo 2. Embora seja aceito atualmente que o modelo 'cascata' em geral não é apropriado para o desenvolvimento de software, a maioria dos processos de desenvolvimento de sistemas é de processos dirigidos a planos que ainda seguem esse modelo.

Figura 10.6 Desenvolvimento de sistemas

Os processos dirigidos a planos são usados na engenharia de sistemas porque diferentes partes do sistema estão sendo desenvolvidas ao mesmo tempo. Para os sistemas que incluem hardware e outros equipamentos, as alterações durante o desenvolvimento podem ser muito caras ou, às vezes, impossíveis. É essencial, portanto, que os requisitos de sistema sejam totalmente compreendidos antes de o desenvolvimento de hardware ou a construção começar. Raramente é possível refazer o projeto do sistema para resolver problemas de hardware. Por essa razão, cada vez mais a funcionalidade de sistema está sendo atribuída ao software do sistema. Isso permite que sejam feitas algumas mudanças durante o desenvolvimento de sistema, em resposta a inevitáveis novos requisitos do sistema.

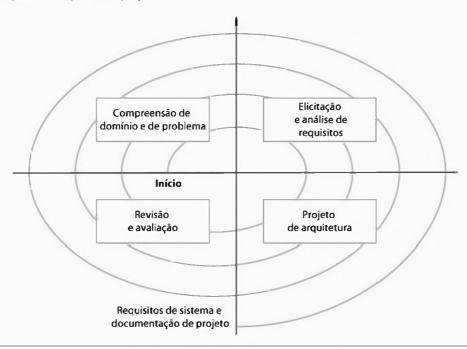
Um dos aspectos mais confusos da engenharia de sistemas é que as empresas usam terminologias diferentes para cada estágio do processo. A estrutura de processo também varia. Às vezes, a engenharia de requisitos é parte do processo de desenvolvimento e, às vezes, é uma atividade separada. No entanto, existem basicamente seis atividades fundamentais no desenvolvimento de sistemas:

1. *Desenvolvimento de requisitos*. Os requisitos de alto nível e requisitos de negócio identificados durante o processo de aquisição precisam ser desenvolvidos em mais detalhes. Os requisitos podem ser atribuídos ao hardware, software ou processos e priorizados para implementação.
2. *Projeto de sistema*. Esse processo coincide significativamente com o processo de desenvolvimento de requisitos. Trata-se de estabelecer a arquitetura global do sistema, identificar os diferentes componentes de sistema e compreender os relacionamentos entre eles.
3. *Engenharia de subsistema*. Esse estágio envolve o desenvolvimento de componentes de software do sistema, a configuração de hardware e software de prateleira, projeto e, se necessário, hardware para fins especiais, além da definição dos processos operacionais para o sistema e o reprojeto de processos essenciais do negócio.
4. *Integração de sistema*. Durante esse estágio, os componentes são colocados juntos para se criar um novo sistema. Só então as propriedades do sistema emergente ficam aparentes.
5. *Teste de sistema*. Geralmente, essa é uma atividade extensiva, prolongada, em que os problemas são descobertos. As fases de engenharia de subsistema e de integração de sistema são reiniciadas para reparar esses problemas, ajustar o desempenho do sistema e implementar novos requisitos. Teste de sistema pode envolver tanto os testes realizados pelo desenvolvedor do sistema quanto os testes de aceitação/usuário pela organização que tenha adquirido o sistema.
6. *Implantação de sistema*. Esse é o processo de tornar o sistema disponível para os usuários, transferir dados dos sistemas existentes e estabelecer comunicações com outros sistemas no ambiente. O processo culmina com um *go live* depois que os usuários começam a usar o sistema para apoiar seu trabalho.

Apesar de todo o processo ser dirigido a planos, os processos de requisitos de desenvolvimento e projeto de sistema estão intimamente ligados. Os requisitos e o projeto de alto nível são desenvolvidos simultaneamente. As restrições impostas por sistemas existentes podem limitar as escolhas de projeto, e essas escolhas podem ser especificadas nos requisitos. Você pode ter de fazer algum projeto inicial para estruturar e organizar o processo de engenharia de requisitos. Como o processo de projeto de sistema continua, você pode descobrir problemas com os requisitos existentes, e podem surgir novos requisitos. Consequentemente, você pode pensar nesses processos ligados como uma espiral, como mostra a Figura 10.7.

A espiral indica que os requisitos afetam as decisões de projeto e vice-versa, e por isso faz sentido intercalar esses processos. Começando no centro, cada volta da espiral pode adicionar detalhes aos requisitos e ao projeto. Algumas voltas podem se concentrar nos requisitos, e outras, no projeto. Às vezes, novos conhecimentos coletados durante os requisitos e o processo de projeto significam que a declaração do problema em si precisa ser mudada.

Para quase todos os sistemas, existem muitos possíveis projetos que atendam aos requisitos. Eles abrangem uma gama de soluções que combinam hardware, software e operações humanas. A solução que você escolhe para o desenvolvimento futuro pode ser a solução técnica mais adequada para atender aos requisitos. No entanto, considerações organizacionais e políticas mais amplas podem influenciar a escolha da solução. Por exemplo, um cliente por parte do governo pode preferir fornecedores nacionais em vez de estrangeiros para seu sistema, mesmo que os produtos nacionais sejam tecnicamente inferiores. Geralmente, essas influências produzem efeitos na fase de revisão e avaliação do modelo em espiral, quando projetos e requisitos podem ser aceitos ou rejeitados. O processo termina quando uma revisão decide que os requisitos e o projeto de alto nível estão suficientemente detalhados para que os subsistemas sejam especificados e projetados.

Figura 10.7 Espiral de requisitos e projeto

Na fase de engenharia de subsistema, os componentes de hardware e software são implementados. Para alguns tipos de sistemas, como naves espaciais, todos os componentes de hardware e software podem ser projetados e construídos durante o processo de desenvolvimento. No entanto, na maioria dos sistemas, alguns componentes são sistemas de prateleira comerciais (COTS). Costuma ser muito mais barato comprar produtos existentes do que desenvolver componentes com propósitos especiais.

Os subsistemas são geralmente desenvolvidos em paralelo. Quando os problemas que atravessam as fronteiras do subsistema são encontrados, uma solicitação de modificação de sistema deve ser feita. Quando os sistemas envolvem engenharia extensiva de hardware, fazer modificações após a fabricação ter sido iniciada mostra-se, em geral, muito caro. Muitas vezes, devem ser encontradas 'soluções' que compensem o problema. Essas 'soluções' geralmente envolvem mudanças de software, em virtude de sua flexibilidade inerente.

Durante a integração de sistemas, você pega os subsistemas desenvolvidos de forma independente e coloca todos juntos para fazer um sistema completo. Essa integração pode ser feita a partir uma abordagem '*big bang*', pela qual todos os subsistemas são integrados ao mesmo tempo. No entanto, por razões técnicas e gerenciais, um processo de integração incremental, em que os subsistemas são integrados um por vez, é a melhor abordagem:

1. Geralmente, é impossível programar o desenvolvimento dos subsistemas de forma que todos terminem ao mesmo tempo.
2. A integração incremental reduz o custo da localização de erros. Se muitos subsistemas estiverem integrados simultaneamente, um erro que surgir durante o teste poderá estar em qualquer um desses subsistemas. Quando um único subsistema é integrado em um sistema já em funcionamento, os erros que ocorrerem provavelmente estarão nos subsistemas recém-integrados ou nas interações entre os subsistemas existentes e o novo subsistema.

Como mais e mais sistemas são construídos por meio da integração de componentes COTS de hardware e software, a distinção entre implementação e integração é cada vez mais tênue. Em alguns casos, não há necessidade de se desenvolver um novo hardware ou software, e a integração é, essencialmente, a fase de implementação do sistema.

O sistema é testado durante e após o processo de integração. Esse teste deve centrar-se em testar as interfaces entre os componentes e o comportamento do sistema como um todo. Inevitavelmente, isso também revelará problemas com subsistemas individuais que precisem ser reparados.

Os defeitos de subsistema que são uma consequência de suposições inválidas sobre outros subsistemas são frequentemente revelados durante a integração do sistema. Isso pode causar disputas entre os contratantes res-

ponsáveis pela implementação de diferentes subsistemas. Quando os problemas são descobertos na interação do subsistema, os contratantes podem argumentar sobre qual subsistema é defeituoso. As negociações sobre como resolver os problemas podem levar semanas ou meses.

A fase final do processo de desenvolvimento é a entrega e implantação de sistema. O software é instalado no hardware e está preparado para operar. Isso pode envolver mais configurações de sistema para refletir o ambiente do local em que este é usado, a transferência de dados dos sistemas existentes, bem como a preparação da documentação e o treinamento de usuários. Nessa fase, você também pode precisar reconfigurar os outros sistemas do ambiente para se assegurar de que o novo sistema interopera com eles.

Embora se trate de uma fase simples, em princípio, muitas dificuldades podem surgir durante a implantação. O ambiente de usuário pode ser diferente daquele previsto pelos desenvolvedores do sistema e adaptar o sistema para lidar com diversos ambientes de usuário pode ser difícil. Os dados existentes podem exigir limpezas extensivas e partes deles podem estar faltando. As interfaces com outros sistemas podem não estar devidamente documentadas.

É óbvia a influência dos processos de desenvolvimento de sistema na proteção e confiança. É durante esses processos que são tomadas decisões sobre os requisitos de confiança e proteção e sobre os compromissos entre custos, cronograma, desempenho e confiança. Os erros humanos em todas as fases do processo de desenvolvimento podem conduzir à introdução de defeitos no sistema; se esses erros ocorrem quando o sistema já está em funcionamento, podem levar à falha de sistema. Os processos de teste e validação são inevitavelmente limitados pelos custos e tempo disponíveis. Como resultado, o sistema não pode ser devidamente testado. Os usuários testam o sistema enquanto ele está sendo usado. Finalmente, os problemas na implantação do sistema podem significar que existe uma incompatibilidade entre o sistema e seu ambiente operacional, os quais podem levar a erros humanos durante o uso do sistema.

10.5 Operação de sistemas

Os processos operacionais são os processos envolvidos no uso do sistema para seus fins definidos. Por exemplo, os operadores de um sistema de controle de tráfego aéreo seguem processos específicos quando as aeronaves entram e saem do espaço aéreo, quando precisam alterar a altura ou a velocidade, quando ocorre uma emergência, e assim por diante. Para novos sistemas, esses processos operacionais precisam ser definidos e documentados durante o processo de desenvolvimento do sistema. Os operadores precisam ser treinados, e outros processos, adaptados para que se faça uso efetivo do novo sistema. Nesse estágio, problemas não detectados podem surgir, pois a especificação de sistema pode conter erros ou omissões. Embora o sistema possa executar a especificação, suas funções podem não atender às reais necessidades operacionais. Assim, os operadores não podem usar o sistema como seus projetistas pretendiam.

O principal benefício de se ter operadores de sistema é que as pessoas têm a capacidade única de responder eficazmente a situações inesperadas, mesmo quando nunca tiveram experiência direta com tais situações. Portanto, quando as coisas dão errado, os operadores frequentemente podem recuperar a situação, embora isso possa significar a violação do processo definido. Os operadores também usam seu conhecimento local para adaptar e melhorar os processos. Normalmente, os processos operacionais reais são diferentes daqueles antecipados pelos projetistas do sistema.

Por conseguinte, você deve projetar processos operacionais para serem flexíveis e adaptáveis. Os processos operacionais não devem ser muito restritivos, não devem exigir que as operações sejam realizadas em determinada ordem e o software de sistema não deve depender de um processo específico a ser seguido. Os operadores geralmente melhoram o processo porque sabem o que funciona ou não em uma situação real.

Um problema que só pode surgir depois que o sistema entra em operação é a operação do novo sistema com os sistemas existentes. Pode haver problemas físicos de incompatibilidade ou pode ser difícil transferir dados de um sistema para outro. Os problemas mais sutis podem surgir porque sistemas diferentes têm diferentes interfaces de usuário. Introduzir um novo sistema pode aumentar a taxa de erro do operador, pois os operadores usam os comandos de interface de usuário para o sistema errado.

10.5.1 Erro humano

Anteriormente, neste capítulo, sugeri que o não determinismo era uma questão importante em sistemas sociotécnicos e que uma razão para isso é que as pessoas no sistema nem sempre se comportam da mesma maneira. Às vezes, as pessoas cometem erros no uso do sistema, e estes podem causar a falha do sistema. Por exemplo, um operador pode se esquecer de registrar que algumas ações foram tomadas e outro operador pode (erroneamente) repetir a ação. Se a ação for de débito ou crédito de uma conta bancária, por exemplo, uma falha de sistema ocorrerá quando o saldo na conta for incorreto.

Como discute Reason (2000), erros humanos sempre ocorrerão. Existem duas maneiras de ver esse problema:

1. *A abordagem de pessoas.* Os erros são considerados de responsabilidade do indivíduo e 'atos inseguros' (como um operador falhar em não participar de uma barreira de segurança) são uma consequência da falta de cuidado individual ou do comportamento imprudente. Pessoas que adotam essa abordagem acreditam que os erros humanos podem ser reduzidos por meio de ameaças de sanções disciplinares, procedimentos mais rigorosos, reciclagem etc. Sua visão é de que o erro é uma falha da pessoa responsável que o cometeu.
2. *A abordagem de sistemas.* O pressuposto básico é de que as pessoas são falíveis e cometem erros. Muitas vezes, os erros que as pessoas cometem são consequência de decisões de projeto de sistema que levam a maneiras erradas de trabalho ou de fatores organizacionais que afetam os operadores do sistema. Bons sistemas devem reconhecer a possibilidade do erro humano e incluir barreiras e salvaguardas que detectem esses erros e permitam que o sistema se recupere antes de a falha ocorrer. Quando uma falha ocorre, o problema não é encontrar uma pessoa para culpar, mas entender como e por que as defesas do sistema não interceptaram o erro.

Eu acredito que a abordagem de sistemas é o caminho certo e que os engenheiros de sistemas devem assumir que os erros humanos ocorrerão durante a operação do sistema. Portanto, para melhorar a proteção e a confiança de um sistema, os projetistas precisam pensar nas defesas e barreiras ao erro humano que devem ser incluídas em um sistema. Eles também devem pensar se essas barreiras devem ser construídas em componentes técnicos do sistema. Se não, elas podem fazer parte dos processos e procedimentos para o uso do sistema ou podem ser diretrizes de operadores que dependam da verificação e julgamento humanos.

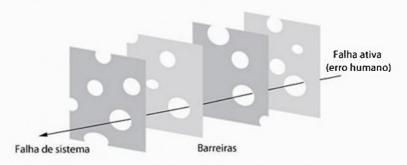
Exemplos de defesas que podem ser incluídas em um sistema são:

1. Um sistema de controle de tráfego aéreo pode incluir um sistema de alerta de conflito automatizado. Quando um controlador instrui uma aeronave para mudar sua velocidade ou altitude, o sistema extrapola sua trajetória para ver se esta pode cruzar com qualquer outra aeronave. Se assim for, soa um alarme.
2. O mesmo sistema pode ter um procedimento claramente definido para gravar as instruções de controle que forem emitidas. Esses procedimentos ajudam a verificar se o controlador emitiu a instrução corretamente e disponibilizou a informação aos outros para a verificação.
3. Controle de tráfego aéreo geralmente envolve uma equipe de controladores que monitoram constantemente o trabalho dos outros. Portanto, quando um erro é cometido, é provável que seja detectado e corrigido antes que ocorra um incidente.

Inevitavelmente, todas as barreiras têm deficiências de algum tipo. Reason as chama de 'condições latentes', pois elas geralmente só contribuem para a falha de sistema quando ocorre algum outro problema. Por exemplo, nas defesas mencionadas anteriormente, uma deficiência de um sistema de alerta é que ele pode levar a muitos alarmes falsos. Os controladores podem, portanto, ignorar os avisos do sistema. Uma deficiência de um sistema procedural pode ser que informações essenciais incomuns podem não ser facilmente registradas. A verificação humana pode falhar quando todas as pessoas envolvidas estão sob estresse e cometem o mesmo erro.

As condições latentes levam à falha do sistema quando as defesas construídas não interceptam uma falha ativa de um operador de sistema. O erro humano é um gatilho para a falha, mas não deve ser considerado o único motivo da falha. Reason explica isso usando seu conhecido modelo 'queijo suíço' de falha do sistema (Figura 10.8).

Nesse modelo, as defesas construídas em um sistema são comparadas a fatias de queijo suíço. Alguns tipos de queijo suíço, como o Emmental, têm buracos, e por isso a analogia é de que as condições latentes são comparáveis com os buracos nas fatias do queijo. A posição desses buracos não é estática, muda dependendo do estado global do sistema sociotécnico. Se cada fatia representa uma barreira, as falhas podem ocorrer quando os furos se alinham ao mesmo tempo, como um erro operacional humano. Uma falha ativa do funcionamento do sistema passa pelos furos e leva a uma falha de sistema global.

Figura 10.8 Modelo queijo suíço de Reason de falha de sistema

Normalmente, é claro, os buracos não devem ser alinhados de modo que as falhas operacionais sejam capturadas pelo sistema. Para reduzir a probabilidade de que falha de sistema resulte de erro humano, os projetistas devem:

1. Projetar um sistema de forma que diferentes tipos de barreiras sejam incluídos. Isso significa que os 'buracos' provavelmente estarão em lugares diferentes e há menores chances de se alinharem os buracos e se falhar em pegar um erro.
2. Minimizar o número de condições latentes em um sistema. Efetivamente, isso significa reduzir o número e o tamanho dos 'buracos' do sistema.

Naturalmente, o projeto do sistema como um todo também deve tentar evitar as falhas ativas que podem provocar a falha do sistema. Isso pode envolver o projeto de processos operacionais e do sistema para garantir que os operadores não estejam sobrecarregados, distraídos ou com quantidades excessivas de informações.

10.5.2 Evolução do sistema

Sistemas complexos de grande porte têm vida útil muito longa. Durante sua vida, eles são alterados para correção dos erros nos requisitos do sistema original e para implementar novos requisitos. Os computadores do sistema tendem a ser substituídos por máquinas novas e mais rápidas. A organização que usa o sistema pode reorganizar-se e, portanto, usar o sistema de uma maneira diferente. O ambiente externo do sistema pode mudar ou forçar mudanças. Portanto, a evolução em que o sistema se altera para acomodar as alterações ambientais é um processo que ocorre junto com os processos normais de operação de sistema. A evolução do sistema envolve reiniciar o processo de desenvolvimento para fazer alterações e extensões para os processos de hardware, software e operacionais do sistema.

A evolução de sistema, tal qual a evolução de software (discutida no Capítulo 9), é inerentemente cara, por vários motivos:

1. As alterações propostas precisam ser analisadas com muito cuidado a partir de uma perspectiva técnica e de negócio. As alterações precisam contribuir com os objetivos do sistema e não devem ser motivadas simplesmente por razões técnicas.
2. Como subsistemas nunca são completamente independentes, as alterações a um subsistema podem afetar negativamente o desempenho ou o comportamento de outros subsistemas. Consequentemente, alterações desses subsistemas podem ser necessárias.
3. Frequentemente, as razões para as decisões do projeto original não são registradas. Os responsáveis pela evolução do sistema precisam compreender por que as decisões de projeto foram tomadas.
4. Enquanto o sistema envelhece, sua estrutura se corrompe pelas alterações e, assim, aumentam-se os custos de novas alterações.

Muitas vezes, os sistemas que evoluíram ao longo do tempo são dependentes de tecnologia de hardware e software obsoleta. Se os sistemas têm um papel fundamental em uma organização, eles são conhecidos como 'sistemas legados'. Geralmente, esses são os sistemas que a organização gostaria de substituir, mas não podem fazê-lo, pois os riscos ou custos de substituição não se justificam.

De uma perspectiva da proteção e confiança, as mudanças nos sistemas geralmente são fontes de problemas e vulnerabilidades. Se as pessoas que implementam a mudança são diferentes daquelas que desenvolveram o sistema, elas podem não ter consciência de que uma decisão de projeto foi tomada por razões de proteção e confiança.

Portanto, elas podem mudar o sistema e perder algumas salvaguardas implementadas deliberadamente quando o sistema foi construído. Além disso, como os testes são muito caros, pode ser impossível efetuar testes completos após cada mudança no sistema. Os efeitos colaterais adversos das mudanças, que introduzem ou expõem defeitos em outros componentes de sistema, podem não ser descobertos.

PONTOS IMPORTANTES

- Sistemas sociotécnicos incluem hardware, software e pessoas. Eles se situam dentro de uma organização e são projetados para apoiar os objetivos e metas organizacionais ou de negócios.
- Fatores humanos e organizacionais, como a estrutura e política organizacional, têm efeito significativo sobre a operação dos sistemas sociotécnicos.
- As propriedades emergentes de um sistema são as características do sistema como um todo, e não de seus componentes. Elas incluem propriedades como desempenho, confiabilidade, usabilidade, segurança e proteção. O sucesso ou fracasso de um sistema é, muitas vezes, dependente dessas propriedades emergentes.
- Os processos fundamentais da engenharia de sistemas são: aquisição de sistema, desenvolvimento de sistema e operação de sistema.
- Aquisição de sistema abrange todas as atividades envolvidas na decisão sobre que sistema comprar e quem deve fornecê-lo. Os requisitos de alto nível são desenvolvidos como parte do processo de aquisição.
- O desenvolvimento de sistema inclui especificação de requisitos, projeto, construção, integração e testes. A integração de sistema, em que os subsistemas de mais de um fornecedor devem ser feitos para trabalharem em conjunto, é particularmente crítica.
- Quando um sistema é colocado em uso, os processos operacionais e o sistema em si precisam mudar para refletir as mudanças nos requisitos de negócio.
- Erros humanos são inevitáveis, e os sistemas devem incluir barreiras para detectar esses erros antes que ocasionem a falha do sistema. O modelo de queijo suíço de Reason explica como o erro humano somado aos defeitos latentes nas barreiras podem levar à falha do sistema.

LEITURA COMPLEMENTAR

'Airport 95: Automated baggage system'. Um estudo de caso excelente e legível sobre o que pode dar errado com um projeto de engenharia de sistemas e como o software tende a ficar com a culpa pelas falhas mais amplas do sistema. (*ACM Software Engineering Notes*, 21 mar. 1996.) Disponível em: <http://doi.acm.org/10.1145/227531.227544>.

'Software system engineering: A tutorial'. Uma boa visão geral da engenharia de sistemas, embora Thayer se concentre exclusivamente em sistemas baseados em computador e não discuta questões sociotécnicas. (THAYER, R. H. *IEEE Computer*, abr. 2002.) Disponível em: <http://dx.doi.org/10.1109/MC.2002.993773>.

Trust in Technology: A Socio-technical Perspective. Esse livro é um conjunto de artigos envolvidos de alguma forma, com a confiança de sistemas sociotécnicos. (CLARKE, K.; HARDSTONE, G.; ROUNCEFIELD, M.; SOMMERVILLE, I. (Orgs.). *Trust in Technology: A Socio-technical Perspective*, Springer, 2006.)

'Fundamentals of Systems Engineering'. Esse é o capítulo introdutório do manual de engenharia de sistemas da NASA. Apresenta uma visão geral do processo de engenharia de sistemas para sistemas espaciais. Embora esses sistemas sejam principalmente técnicos, existem questões sociotécnicas a serem consideradas. Obviamente, a confiança é muito importante. (In: *Nasa Systems Engineering Handbook*, NASA-SP2007-6105, 2007.) Disponível em: <http://education.ksc.nasa.gov/esmdspacegrant/Documents/NASA%20SP-2007-6105% 20Rev% 201% 20final % 2031Dec2007.pdf>.

EXERCÍCIOS

10.1 Dê dois exemplos de funções de governo apoiadas por sistemas sociotécnicos complexos e explique por que, em um futuro próximo, essas funções não poderão ser totalmente automatizadas.

10.2 Explique por que o ambiente no qual um sistema baseado em computador é instalado pode ter efeitos imprevistos capazes de conduzir à falha de sistema. Ilustre sua resposta com um exemplo diferente do usado neste capítulo.

10.3 Por que é impossível inferir as propriedades emergentes de um sistema complexo a partir das propriedades dos componentes de sistema?

10.4 Por que, em alguns casos, é difícil decidir se aconteceu ou não uma falha em um sistema sociotécnico? Ilustre sua resposta com exemplos do MHC-PMS, discutido nos capítulos anteriores.

10.5 O que é um 'problema severo'? Explique por que o desenvolvimento de um sistema nacional de registros médicos deve ser considerado um 'problema severo'.

10.6 Um sistema multimídia de museu virtual, que oferece experiências virtuais da Grécia Antiga, está sendo desenvolvido por um consórcio de museus europeus. O sistema deve oferecer aos usuários o recurso de visualizar os modelos 3D da Grécia Antiga por meio de um *browser*-padrão, e também deve suportar uma experiência de realidade virtual imersiva. Quais dificuldades políticas e organizacionais podem surgir quando o sistema for instalado nos museus que compõem o consórcio?

10.7 Por que a integração de sistema é uma parte particularmente importante do processo de desenvolvimento de sistemas? Sugira três questões sociotécnicas que podem causar dificuldades no processo de integração de sistema.

10.8 Explique por que os sistemas legados podem ser críticos para a operação de um negócio.

10.9 Quais são os argumentos a favor e contra a engenharia de sistemas ser considerada uma profissão de direito próprio, tal qual a engenharia elétrica ou a engenharia de software?

10.10 Você é um engenheiro envolvido no desenvolvimento de um sistema financeiro. Durante a instalação, você descobre que esse sistema fará com que um significativo número de pessoas se torne redundante. As pessoas no ambiente negam-lhe o acesso a informações essenciais para completar a instalação do sistema. Até que ponto você deve, como um engenheiro de sistemas, envolver-se nessa situação? É de sua responsabilidade profissional concluir a instalação, conforme o contrato? Você deveria simplesmente abandonar o trabalho até que a organização contratante resolva o problema?

REFERÊNCIAS

ACKROYD, S.; HARPER, R.; HUGHES, J. A.; SHAPIRO, D. *Information Technology and Practical Police Work*. Milton Keynes: Open University Press, 1992.

ANDERSON, R. J.; HUGHES, J. A.; SHARROCK, W. W. *Working for Profit*: The Social Organization of Calculability in an Entrepreneurial Firm. Aldershot: Avebury, 1998.

CHECKLAND, P. *Systems Thinking, Systems Practice*. Chichester: John Wiley & Sons, 1981.

CHECKLAND, P.; SCHOLES, J. *Soft Systems Methodology in Action*. Chichester: John Wiley & Sons, 1990.

MUMFORD, E. User Participation in a Changing Environment—Why we need it. In: KNIGHT, K. (Org.). *Participation in Systems Development*. Londres: Kogan Page, 1989.

REASON, J. Human error: Models and management. *British Medical* J., v. 320, 2000, p. 768-770.

RITTEL, H.; WEBBER, M. Dilemmas in a General Theory of Planning. *Policy Sciences*, v. 4, 1973, p. 155-169.

STEVENS, R.; BROOK, P.; JACKSON, K.; ARNOLD, S. *Systems Engineering*: Coping with Complexity. Londres: Prentice Hall, 1998.

SUCHMAN, L. *Plans and situated actions*: the problem of human-machine communication. Nova York: Cambridge University Press, 1987.

SWARTZ, A. J. Airport 95: Automated Baggage System? *ACM Software Engineering Notes*, v. 21, n. 2, 1996, p. 79-83.

THAYER, R. H. Software System Engineering: A Tutorial. *IEEE Computer*, v. 35, n. 4, 2002, p. 68-73.

THOMÉ, B. *Systems Engineering: Principles and Practice of Computer-based Systems Engineering*. Chichester: John Wiley & Sons, 1993.

WHITE, S.; ALFORD, M.; HOLTZMAN, J.; KUEHL, S.; McCAY, B.; OLIVER, D.; OWENS, D.; TULLY. C.; WILLEY, A. *Systems Engineering of Computer-Based Systems*. *IEEE Computer*, v. 26, n. 11, 1993, p. 54-65.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 **11** 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Confiança e proteção

Objetivos

O objetivo deste capítulo é apresentar a confiança e a proteção de software. Com a leitura deste capítulo, você:

- entenderá por que, geralmente, a confiança e a proteção são mais importantes do que as características funcionais de um sistema de software;
- entenderá as quatro principais dimensões da confiança, ou seja, disponibilidade, confiabilidade, segurança e proteção;
- estará ciente da terminologia especializada quando se discute proteção e confiança;
- entenderá que, para alcançar um software protegido e confiável, é preciso evitar erros durante o desenvolvimento de um sistema, detectar e remover os erros enquanto o sistema está em uso e limitar os danos causados por falhas operacionais.

Conteúdo

Como os sistemas computacionais estão profundamente enraizados em nossos negócios e vidas pessoais, estão aumentando os problemas que resultam dos sistemas e falhas de software. Uma falha do software de servidor em uma empresa de comércio eletrônico pode causar uma grande perda de receita e, inclusive, à perda dos clientes da empresa. Um erro de software em um sistema de controle embutido em um carro pode levar a *recalls* daquele modelo para reparação e, na pior das hipóteses, pode ser um fator de causa de acidentes. A infecção de PCs de uma empresa com *malwares* pode resultar na perda ou em danos a informações confidenciais, e requer operações de limpeza de alto custo para resolver o problema.

Como os sistemas intensivos de software são tão importantes para os governos, empresas e indivíduos, é essencial que o software usado seja confiável. O software deve estar disponível quando necessário e deve funcionar corretamente e sem efeitos colaterais indesejáveis, como a divulgação de informações não autorizadas. O termo 'confiança' foi proposto por Laprie (1995) para cobrir os sistemas relacionados com atributos de disponibilidade, confiabilidade, segurança e proteção. Conforme discuto na Seção 11.1, essas propriedades estão intimamente ligadas, portanto, faz sentido usar um único termo para traduzi-las.

Ultimamente, a confiança dos sistemas costuma ser mais importante do que sua funcionalidade detalhada. As razões para isso são:

1. *Falhas de sistema afetam um grande número de pessoas.* Muitos sistemas incluem funções raramente usadas. Se essas funções ficarem de fora do sistema, apenas um pequeno número de usuários será afetado. As falhas de sistema, que afetam a disponibilidade de um sistema, por sua vez, potencialmente atingem todos os seus usuários. Falhas podem significar que negócios normais são impossíveis.
2. *Usuários muitas vezes rejeitam sistemas não confiáveis, inseguros ou não protegidos.* Se o usuário perceber que um sistema é não confiável ou não protegido, ele se recusará a usá-lo. Além disso, ele também pode se recusar a com-

prar ou a usar outros produtos da mesma empresa que produziu o sistema não confiável, pois acredita que esses produtos também podem ser pouco confiáveis ou não protegidos.

3. *Custos de falha de sistema podem ser enormes.* Para algumas aplicações, como um sistema de controle de reator ou um sistema de navegação de aeronave, o custo de falha de sistema é de magnitude maior do que o custo do sistema de controle.
4. *Sistemas não confiáveis podem causar perda de informações.* A coleta e a manutenção de dados são procedimentos muito caros; geralmente, os dados valem muito mais do que o sistema em que são processados. Os custos de recuperação de dados perdidos ou corrompidos costumam ser muito elevados.

Como discutido no Capítulo 10, o software sempre faz parte de um sistema mais amplo. Ele executa em um ambiente operacional que inclui o hardware no qual o software é executado, os usuários do software e os processos organizacionais ou de negócios em que o software é usado. Portanto, ao projetar um sistema confiável, você precisa considerar:

1. *Falha de hardware.* As falhas no hardware de sistema podem acontecer por erros em seu projeto, por falhas na fabricação dos componentes ou porque os componentes chegaram ao fim de sua vida natural.
2. *Falha de software.* O sistema de software pode falhar devido a erros em suas especificações, projeto ou implementação.
3. *Falha operacional.* Os usuários podem falhar na tentativa de usar ou operar o sistema corretamente. Como o hardware e o software se tornaram mais confiáveis, falhas na operação são, talvez, a maior causa de falhas de sistema.

Geralmente, essas falhas são inter-relacionadas. Um componente de hardware falho pode significar que os operadores de sistema precisam lidar com uma situação inesperada, além de carga de trabalho adicional. Tal situação gera estresse, e pessoas estressadas muitas vezes cometem erros. Isso pode causar falhas no software, o que significa mais trabalho para os operadores, ainda mais estresse, e assim por diante.

Como resultado, é particularmente importante que os projetistas de sistemas intensivos de software confiáveis tenham uma visão holística de sistemas e não se concentrem em um único aspecto destes, como o software ou o hardware. Se o hardware, o software e os processos operacionais forem projetados separadamente, sem levar em conta os potenciais pontos fracos de outras partes do sistema, então é mais provável que os erros ocorram nas interfaces entre as diferentes partes do sistema.

11.1 Propriedades da confiança

Todos nós estamos familiarizados com falhas nos sistemas computacionais. Às vezes, sem qualquer razão aparente, nossos computadores falham ou erram de alguma forma. Programas executados nesses computadores podem não funcionar como o esperado e, ocasionalmente, podem corromper os dados gerenciados pelo sistema. Aprendemos a conviver com essas falhas, mas poucos de nós confiam completamente nos computadores pessoais, usadas normalmente.

A confiança de um sistema de computador é uma propriedade do sistema que reflete sua fidedignidade. Fidedignidade aqui significa, essencialmente, o grau de confiança de um usuário no funcionamento esperado pelo sistema, no fato de que o sistema não 'falhará' em condições normais de uso. Não faz sentido expressar essa confiança numericamente. Em vez disso, usamos termos relativos, como 'não confiável', 'muito confiável' e 'ultraconfiável' para refletir o grau de confiança que podemos ter em um sistema.

Certamente, confiabilidade e utilidade não são sinônimos. Eu não acho que o processador de texto que usei para escrever este livro seja um sistema muito confiável. Às vezes, ele congela e precisa ser reiniciado. Mas, porque é muito útil, estou disposto a tolerar uma falha ocasional. No entanto, como reflexo de minha desconfiança, prefiro salvar meu trabalho com frequência e manter múltiplas cópias de *backup*. Eu compenso a falta de confiança de sistema por meio de ações que limitam os danos que poderiam resultar na falha de sistema.

Como mostrado na Figura 11.1, existem quatro dimensões principais de confiança:

1. *Disponibilidade.* Informalmente, a disponibilidade de um sistema é a probabilidade de, a qualquer instante, ele estar ativo, funcionando e ser capaz de prestar serviços úteis aos usuários.
2. *Confiabilidade.* Informalmente, a confiabilidade de um sistema é a probabilidade de, durante determinado período, o sistema prestar serviços corretamente, conforme o esperado pelo usuário.
3. *Segurança.* Informalmente, a segurança de um sistema é a análise da probabilidade de o sistema causar danos às pessoas ou a seu ambiente.

Figura 11.1 Principais propriedades da confiança

4. *Proteção*. Informalmente, a proteção de um sistema é uma análise da probabilidade de ele resistir às invasões acidentais ou deliberadas.

As propriedades de confiança mostradas na Figura 11.1 são propriedades complexas que podem ser decompostas em uma série de outras mais simples. Por exemplo, proteção inclui 'integridade' (garantindo que o programa e os dados do sistema não estejam danificados) e 'confidencialidade' (garantindo que as informações só podem ser acessadas por pessoas autorizadas). Confiabilidade inclui 'correção' (garantia de que os serviços de sistema são conforme diz suas especificações), 'precisão' (garantia de que a informação é entregue com o nível de detalhamento adequado) e 'em tempo certo' (garantindo que a informação é entregue quando necessário).

Compreende-se que essas propriedades da confiança não são aplicáveis a todos os sistemas. Para o sistema da bomba de insulina, apresentado no Capítulo 1, as propriedades mais importantes são a disponibilidade (que deve trabalhar quando necessário), a confiabilidade (que deve administrar a dose correta de insulina) e a segurança (que nunca deve fornecer uma dose perigosa de insulina). A proteção não é um problema, pois a bomba não precisa manter informações confidenciais. Ela não fica em rede e, portanto, não pode ser maliciosamente atacada. Para o sistema meteorológico no deserto, a disponibilidade e a confiabilidade são as propriedades mais importantes, pois os custos de reparo podem ser muito altos. Para o sistema de informações de pacientes, a proteção é particularmente importante, em virtude dos dados privados sensíveis que são mantidos.

Assim como essas quatro propriedades principais da confiança, você também pode pensar em outras propriedades do sistema como propriedades da confiança:

1. *Reparabilidade*. Falhas do sistema são inevitáveis, mas a interrupção causada pela falha pode ser minimizada se o sistema puder ser reparado rapidamente. Para que isso aconteça, deve ser possível diagnosticar o problema, acessar o componente que falhou e fazer alterações para corrigir esse componente. Em software, a reparabilidade é aprimorada quando a organização que usa o sistema tem acesso ao código-fonte e tem as habilidades para fazer alterações necessárias. Softwares *open source* tornam isso mais fácil, mas o reúso dos componentes pode tornar esse processo mais difícil.
2. *Manutenibilidade*. Enquanto os sistemas são usados, novos requisitos surgem e, para manter a utilidade de um sistema, é importante fazer modificações para acomodar esses novos requisitos. Software manutenível é um software que pode ser adaptado economicamente para lidar com novos requisitos, em que existe uma baixa probabilidade de introdução de novos erros no sistema em virtude de mudanças nele.
3. *Capacidade de sobrevivência*. Um atributo muito importante para os sistemas baseados na Internet é a capacidade de sobrevivência (ELLISON et al., 1999b). 'Sobrevivência' é a capacidade de um sistema de continuar prestando serviço mesmo sob ataque e, potencialmente, enquanto parte do sistema é desativada. Os trabalhos sobre sobrevivência concentram-se em identificar os principais componentes de sistema e em garantir que estes possam prestar serviço mesmo que minimamente. Três estratégias são usadas para aumentar a capacidade de sobrevivência — resistência a ataques; reconhecimento de ataque; e recuperação de danos causados por um ataque (ELLISON et al., 1999a; ELLISON et al., 2002). No Capítulo 14, discuto esse assunto em detalhes.
4. *Tolerância a erros*. Essa propriedade pode ser considerada parte da usabilidade, e reflete o grau em que o sistema foi projetado de modo a evitar e tolerar erros de entradas de usuário. Quando ocorrem erros de usuário, o

sistema deve, na medida do possível, detectar esses erros e, então, corrigi-los automaticamente ou solicitar ao usuário a reentrada de seus dados.

A noção de confiança de sistema como uma propriedade abrangente foi desenvolvida porque as propriedades de confiança, disponibilidade, proteção, confiabilidade e segurança estão intimamente relacionadas. Geralmente, a operação segura de um sistema depende de ele estar disponível e operando de forma confiável. Um sistema pode tornar-se não confiável porque um intruso corrompeu seus dados. Ataques de negação de serviço em um sistema destinam-se a comprometer sua disponibilidade. Se um sistema é infectado por um vírus, sua confiabilidade e segurança ficam abaladas, pois o vírus pode mudar seu comportamento.

Portanto, para desenvolver um software confiável, você precisa garantir que:

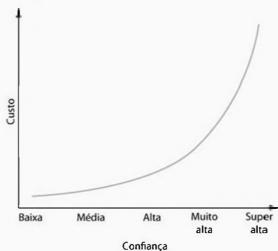
1. Seja evitada a introdução de erros acidentais no sistema durante a especificação e o desenvolvimento de software.
2. Sejam projetados processos de verificação e validação, eficazes na descoberta de erros residuais que afetam a confiança do sistema.
3. Sejam projetados mecanismos de proteção que protejam contra ataques externos capazes de comprometer a disponibilidade ou a proteção do sistema.
4. O sistema implantado e seu software de suporte sejam configurados corretamente para seu ambiente operacional.

Além disso, você deve assumir que o software não é perfeito e que as falhas de software podem ocorrer. Seu sistema deve, portanto, incluir mecanismos de recuperação que tornem possível a restauração do serviço normal do sistema o mais rapidamente possível.

A necessidade de tolerância a defeitos significa que os sistemas confiáveis precisam incluir códigos redundantes para ajudá-los a se monitorar, a detectar estados errôneos e a se recuperar de defeitos antes que ocorram falhas. Isso afeta o desempenho dos sistemas, pois verificações adicionais são necessárias a cada vez que o sistema funciona. Por isso, os projetistas geralmente precisam trocar o desempenho pela confiança. Pode ser necessário não fazer *check out* do sistema, porque eles tornam o sistema lento. No entanto, isso implica risco, pois algumas falhas ocorrem devido a um defeito torna o sistema lento .

Por causa dos custos extras com projeto, implementação e validação, aumentar a confiança de um sistema amplia significativamente seus custos de desenvolvimento. Em particular, os custos de validação são altos para sistemas que devem ser ultraconfiáveis, como sistemas críticos de controle de segurança. Além de validar se o sistema atende aos seus requisitos, o processo de validação pode precisar provar para um regulador externo que o sistema é seguro. Por exemplo, sistemas de aeronaves precisam demonstrar aos órgãos reguladores, como a Autoridade Federal de Aviação, que a probabilidade de uma falha catastrófica de sistema que afete a segurança de uma aeronave é extremamente baixa.

A Figura 11.2 mostra o relacionamento entre os custos e as melhorias incrementais em confiança. Se o software não é muito confiável, você pode obter melhorias significativas com custos relativamente baixos, usando melhor

Figura 11.2 Curva de custo/confiança

a engenharia de software. No entanto, se você já estiver usando boas práticas, os custos de melhoria são muito maiores e os benefícios de melhoria são menores. Existe ainda o problema de testar seu software para demonstrar que ele é confiável. Isso depende da execução de muitos testes e da análise do número de falhas ocorridas. Conforme o software se torna mais confiável, o número de falhas diminui. Consequentemente, mais e mais testes são necessários para tentar avaliar quantos problemas permanecem nele. Como o teste é muito caro, isso aumenta bastante o custo de sistemas de alta confiabilidade.

11.2 Disponibilidade e confiabilidade

A disponibilidade e a confiabilidade de sistema são propriedades intimamente relacionadas, que podem ser expressas em probabilidades numéricas. A disponibilidade de um sistema é a probabilidade de o sistema estar ativo e funcionando para fornecer serviços aos usuários quando estes solicitarem. A confiabilidade de um sistema é a probabilidade de os serviços do sistema serem entregues, tal como definido na especificação de sistema. Se, em média, duas entradas em cada mil provocam falhas, a confiabilidade, expressa como uma taxa de ocorrência de falha, é de 0,002. Se a disponibilidade é 0,999, isso significa que, durante algum período, o sistema está disponível para 99,9% desse tempo.

A confiabilidade e a disponibilidade estão intimamente relacionadas, mas às vezes uma é mais importante que a outra. Se os usuários esperam serviços de um sistema de forma contínua, então o sistema tem um requisito de alta disponibilidade, e deve estar disponível sempre que uma solicitação é feita. No entanto, se as perdas que resultam de uma falha de sistema forem baixas, e o sistema puder se recuperar rapidamente, as falhas não afetarão seriamente os usuários do sistema. Nesses sistemas, os requisitos de confiabilidade podem ser relativamente baixos.

Um comutador de telefones que cria rotas de chamadas telefônicas é um exemplo de um sistema no qual a disponibilidade é mais importante do que a confiabilidade. Os usuários esperam um tom de discagem quando pegam um telefone, logo, o sistema tem requisitos de alta disponibilidade. Se uma falha do sistema ocorre quando uma conexão está sendo criada, esta em geral se recupera rapidamente. Normalmente, os comutadores podem reiniciar o sistema e refazer a tentativa de conexão. Isso pode ocorrer muito rapidamente, e os usuários de telefone podem sequer observar que ocorreu uma falha. Além disso, mesmo quando uma chamada é interrompida, as consequências geralmente não são graves. Portanto, a disponibilidade, em vez de confiabilidade, é um requisito-chave de confiança para esse tipo de sistema.

A confiabilidade e a disponibilidade de sistema podem ser definidas mais precisamente como segue:

1. *Confiabilidade.* É a probabilidade de uma operação livre de falhas durante um tempo especificado, em determinado ambiente, para uma finalidade específica.
2. *Disponibilidade.* É a probabilidade de um sistema, em determinado momento, ser operacional e capaz de entregar os serviços solicitados.

Um dos problemas práticos no desenvolvimento de sistemas confiáveis é que nossas noções intuitivas de confiabilidade e disponibilidade são, por vezes, mais amplas do que essas definições limitadas. A definição de confiabilidade estabelece que o ambiente em que o sistema é usado e a finalidade para qual é usado devem ser levados em conta. Se a confiabilidade de sistema em um ambiente for mensurada, não se pode presumir que será a mesma se o sistema for usado de uma maneira diferente.

Por exemplo, digamos que você meça a confiabilidade de um processador de texto em um ambiente de escritório, onde a maioria dos usuários não está interessada no funcionamento do software. Eles seguem as instruções para seu uso e não tentam fazer experiências com o sistema. Se você medir a confiabilidade do mesmo sistema em um ambiente universitário, a confiabilidade pode ser bem diferente. Aqui, os alunos podem explorar os limites do sistema e usá-lo de formas inesperadas, o que pode resultar em falhas de sistema que não ocorreriam no ambiente mais restrito de escritório.

Essas definições padronizadas de disponibilidade e confiabilidade não levam em conta a gravidade da falha ou as consequências da indisponibilidade. Frequentemente, as pessoas aceitam falhas menores de sistema, mas estão muito preocupadas com falhas graves, com elevados custos resultantes. Por exemplo, falhas de computador que corrompem os dados armazenados são menos aceitáveis do que falhas que congelam a máquina e que podem ser resolvidas reiniciando-se o computador.

Uma definição estrita de confiabilidade relaciona a implementação do sistema com suas especificações. Ou seja, o sistema está se comportando de forma confiável se seu comportamento é coerente com o definido na

especificação. No entanto, uma causa comum de não confiabilidade é a não correspondência da especificação do sistema às expectativas de seus usuários. Infelizmente, muitas especificações estão incompletas ou incorretas, e cabe aos engenheiros de software interpretar como o sistema deve se comportar. Como eles não são especialistas em domínio, não podem, portanto, implementar o comportamento que os usuários esperam. Também é verdade, é claro, que os usuários não leem as especificações de sistema. Portanto, podem ter expectativas irrealistas com relação ao sistema.

A disponibilidade e a confiabilidade são, obviamente, ligadas, assim como falhas no sistema podem travá-lo. No entanto, a disponibilidade não depende apenas do número de falhas no sistema, mas também do tempo necessário para reparar os defeitos que causaram a falha. Portanto, se o sistema A falha uma vez por ano e o sistema B falha uma vez por mês, então A é claramente mais confiável do que B. No entanto, suponha que o sistema A demore três dias para reiniciar após uma falha, enquanto o sistema B leve dez minutos. A disponibilidade do sistema B ao ano (120 minutos de tempo ocioso) é muito melhor do que a do sistema A (4.320 minutos de tempo ocioso).

A interrupção causada pelos sistemas indisponíveis não se reflete na métrica simples que especifica a porcentagem de tempo que o sistema está disponível. O momento em que o sistema falha também é significativo. Se um sistema está indisponível uma hora por dia, entre 3 e 4 horas da manhã, isso não afetará muitos usuários. No entanto, se o mesmo sistema estiver indisponível por dez minutos durante o dia de trabalho, a indisponibilidade de sistema provavelmente terá um efeito muito maior.

Problemas de confiabilidade e disponibilidade do sistema são geralmente causados por falhas de sistema. Algumas dessas falhas são uma consequência de erros de especificação ou falhas em outros sistemas relacionados, como sistemas de comunicações. No entanto, muitas falhas são uma consequência de comportamentos errados do sistema, resultantes de defeitos nele. Quando se discute a confiabilidade, é útil usar a terminologia precisa e distinguir entre os termos 'defeito', 'erro' e 'falha'. Na Tabela 11.1, eu defino esses termos e ilustro cada definição com um exemplo do sistema meteorológico no deserto.

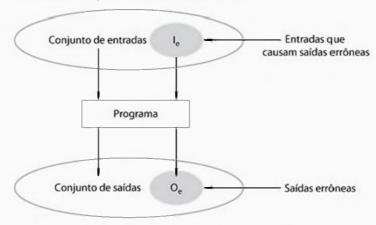
Quando uma entrada ou uma sequência de entradas gera códigos defeituosos em um sistema para ser executado, cria-se um estado errado que pode levar a uma falha de software. A Figura 11.3, derivada de Littlewood (1990), mostra um sistema de software como um mapeamento de um conjunto de entradas para um conjunto de saídas. Dada uma sequência de entradas, o programa responde produzindo uma saída correspondente. Por exemplo, dada uma entrada de uma URL, um *browser* Web produz uma saída, a exibição da página Web solicitada.

A maioria das entradas não leva à falha de sistema. No entanto, algumas entradas ou combinações de entradas, mostradas na elipse sombreada I_e da Figura 11.3, causam falhas de sistema ou geram saídas errôneas. A confiabilidade do programa depende do número de entradas de sistema que são parte do conjunto de entradas que levam a uma saída errônea. Se as entradas do conjunto I_e forem executadas por partes do sistema frequentemente usadas, as falhas serão frequentes. No entanto, se as entradas em I_e forem executadas por um código que raramente é usado, os usuários dificilmente perceberão as falhas.

Como cada usuário usa o sistema de maneiras diferentes, eles têm diferentes percepções de sua confiabilidade. Os defeitos que afetam a confiabilidade do sistema para um usuário podem nunca ser revelados no modo de trabalho de outra pessoa (Figura 11.4). Na Figura 11.4, o conjunto de entradas errôneas corresponde à elipse rotulada como I_e na Figura 11.3. O conjunto de entradas produzidas pelo Usuário 2 cruza com o conjunto de entradas errôneas. Portanto, o Usuário 2 experimentará algumas falhas de sistema. No entanto, o Usuário 1 e o Usuário 3 nunca usam entradas do conjunto errôneo. Para eles, o software será sempre confiável.

Tabela 11.1 Terminologia de confiabilidade

Termo	Descrição
Erro humano ou engano	O comportamento humano, que resulta na introdução de defeitos em um sistema. Por exemplo, no sistema meteorológico no deserto, um programador pode decidir que a forma de calcular o tempo para a próxima transmissão é acrescentar uma hora à hora atual. Isso funciona, exceto quando o tempo de transmissão é entre as 23:00 hs e a meia-noite (meia-noite é 00:00 no horário de 24 horas).
Defeito de sistema	Uma característica de um sistema de software que pode levar a um erro de sistema. O defeito é a inclusão do código para adicionar uma hora à hora da última transmissão, sem verificar se já passou das 23:00 hs.
Erro de sistema	Um estado errôneo de sistema que pode levar a um comportamento do sistema inesperado por seus usuários. O valor do tempo de transmissão é definido incorretamente (a 24.XX em vez de 00.XX) quando o código com defeito é executado.
Falha de sistema	Um evento que ocorre em algum momento em que o sistema não fornece um serviço como esperado por seus usuários. Nenhum dado meteorológico é transmitido porque a hora é inválida.

Figura 11.3 Um sistema como um mapeamento de entradas/saídas

A confiabilidade prática de um programa depende do número de entradas que causam saídas errôneas (falhas) durante o uso normal do sistema pela maioria dos usuários. Os defeitos de software que só ocorrem em situações excepcionais têm pouco efeito prático sobre a confiabilidade do sistema. Consequentemente, a remoção de defeitos de software pode não melhorar significativamente sua confiabilidade geral. Mills et al. (1987) descobriram que a remoção de 60% dos erros conhecidos em seu software levou a uma melhoria da confiabilidade em 3%. Adams (1984), em um estudo de produtos de software da IBM, observou que muitos defeitos nos produtos só eram passíveis de causar falhas após centenas ou milhares de meses de uso do produto.

Defeitos de sistema nem sempre resultam em erros de sistema, e erros de sistema não resultam necessariamente em falhas de sistema. As razões para isso são as seguintes:

1. Nem todos os códigos de um programa são executados. O código que inclui um defeito (por exemplo, a falha para iniciar uma variável) pode nunca ser executado em virtude da maneira como o software é usado.
2. Os erros são transitórios. A variável de estado pode ter um valor incorreto, causado pela execução de um código defeituoso. Entretanto, antes disso, ela pode ser acessada e uma falha de sistema pode ser provocada; alguma outra entrada de sistema pode ser processada e o estado para um valor válido pode ser redefinido.
3. O sistema pode incluir a detecção de defeitos e mecanismos de proteção. Isso assegura que o comportamento errôneo seja descoberto e corrigido antes que os serviços do sistema sejam afetados.

Outra razão pela qual os defeitos em um sistema podem não levar à falha de sistema é que, na prática, os usuários adaptam seu comportamento para evitar entradas que eles sabem que causam falhas no programa. Os usuários experientes 'contornam' os recursos do software que eles sabem que não são confiáveis. Por exemplo, no editor de texto que usei para escrever este livro, eu evito certos recursos, como numeração automática. Quando usei autonumeração, muitas vezes deu errado. Reparar os defeitos de recursos não usados não faz diferença prática

Figura 11.4 Padrões de uso do software

para a confiabilidade de sistema. Os usuários compartilham informações sobre os problemas e como os contornar, de modo que os efeitos dos problemas de um software são reduzidos.

A distinção entre defeitos, erros e falhas, explicada na Tabela 11.1, ajuda a identificar três abordagens complementares usadas para melhorar a confiabilidade de um sistema:

1. *Prevenção de defeitos.* Técnicas de desenvolvimento são usadas para minimizar a possibilidade de erros humanos e/ou enganos antes que eles resultem na introdução de defeitos de sistema. Exemplos dessas técnicas incluem evitar construções de linguagem de programação propensas a erro, como ponteiros e uso da análise estática para detectar anomalias de programa.
2. *Detecção e remoção de defeitos.* O uso de técnicas de verificação e validação aumenta as chances de detecção e remoção de defeitos antes de o sistema ser usado. Testes e depuração sistemáticos são exemplos de técnicas de detecção de defeitos.
3. *Tolerância a defeitos.* São as técnicas que asseguram que os defeitos em um sistema não resultam em erros de sistema ou que os erros de sistema não resultam em falhas de sistema. A incorporação de recursos de autoverificação em um sistema e o uso de módulos redundantes de sistemas são exemplos de técnicas de tolerância a defeitos.

A aplicação prática dessas técnicas é discutida no Capítulo 13, que abrange técnicas de engenharia de software confiável.

11.3 Segurança

Os sistemas críticos de segurança são os sistemas nos quais é essencial que a operação de sistema seja sempre segura, ou seja, que o sistema nunca deve causar danos às pessoas ou ao ambiente, mesmo que ocorra uma falha. Exemplos de sistemas críticos de segurança incluem sistemas de controle e monitoramento de aeronaves, sistemas de controle de processos de plantas químicas e farmacêuticas e sistemas de controle de automóvel.

O controle de hardware de sistemas críticos de segurança é mais simples de implementar e analisar do que o controle de software. Ultimamente, entretanto, construímos sistemas de tal complexidade que não podem mais ser controlados pelo hardware sozinho. Um controle de software é essencial pela necessidade de gerenciar um grande número de sensores e atuadores com leis de controle complexas. Por exemplo, as aeronaves militares avançadas são aerodinamicamente instáveis e requerem ajuste contínuo controlado por software de seu voo para garantir que não ocorram acidentes.

O software crítico de segurança divide-se em duas classes:

1. *Software crítico de segurança primária.* Esse é um software embutido como um controlador em um sistema. O mau funcionamento do software pode causar mau funcionamento do hardware, o que resulta em danos às pessoas ou ao ambiente. O software de bomba de insulina, apresentado no Capítulo 1, é um exemplo de um software crítico primário de segurança. Falha de sistema pode ocasionar danos aos usuários.
2. *Software crítico de segurança secundária.* Esse é um software que pode resultar indiretamente em um dano. Um exemplo desse tipo de software é um sistema de projeto de engenharia auxiliado por computador cujo mau funcionamento pode resultar em um defeito de projeto no objeto que esteja sendo projetado. Esse defeito pode causar danos às pessoas se o sistema projetado tiver mau funcionamento. Outro exemplo de um sistema crítico de segurança secundária é o sistema de gerenciamento de saúde mental, MHC-PMS. Uma falha nesse sistema, no qual um paciente instável não pode ser tratado adequadamente, pode levar os pacientes a se ferirem ou ferirem a outros.

A confiabilidade e a segurança de um sistema estão relacionadas, mas um sistema confiável pode ser inseguro e vice-versa. O software pode, ainda, comportar-se de tal forma que o resultado do comportamento do sistema cause um acidente. Existem quatro razões pelas quais os sistemas de software que são confiáveis não são necessariamente seguros:

1. Nós nunca podemos estar 100% certos de que um sistema de software seja livre de defeitos ou tolerante a defeitos. Defeitos não detectados podem ficar adormecidos por um longo tempo e falhas de software podem ocorrer após vários anos de funcionamento confiável.
2. A especificação pode ser incompleta, sem a descrição do comportamento requerido do sistema em algumas situações críticas. Uma elevada porcentagem de mau funcionamentos de sistema (BOEHM et al., 1975; ENDRES,

1975; LUTZ, 1993; NAKAJO e KUME, 1991) resulta da especificação, e não de erros de projeto. Em um estudo de erros em sistemas embutidos, Lutz conclui:

> *... dificuldades com os requisitos são a causa fundamental de erros relacionados à segurança de software, os quais foram inseridos até a integração e teste de sistema.*

3. Maus funcionamentos de hardware podem levar o sistema a se comportar de forma imprevisível, bem como apresentar o software com um ambiente imprevisto. Quando componentes estão perto de falha física, eles podem se comportar de forma errática e gerar sinais que estão fora dos intervalos que podem ser manipulados pelo software.
4. Os operadores de sistema podem gerar entradas que por si só não são erradas, mas, em algumas situações, podem causar um mau funcionamento de sistema. Um exemplo engraçado ocorreu quando o trem de pouso de uma aeronave entrou em colapso enquanto estava no chão. Aparentemente, um técnico apertou um botão que instruiu o software de gerenciamento a levantar o trem de pouso. O software realizou perfeitamente a instrução do mecânico. No entanto, o sistema não deveria ter permitido o comando a menos que o avião estivesse no ar.

Um vocabulário especializado para discutir sistemas críticos de segurança foi criado, e é importante entender os termos específicos usados. A Tabela 11.2 resume algumas definições de termos importantes, com exemplos extraídos do sistema de bomba de insulina.

A chave para garantir a segurança é assegurar que os acidentes não ocorram e/ou que as consequências de um acidente sejam mínimas. Isso pode ser alcançado de três maneiras complementares:

1. *Prevenção de perigos.* O sistema é projetado de modo que os riscos sejam evitados. Por exemplo, um sistema de corte que exige que um operador use as duas mãos para apertar botões separados simultaneamente evita o perigo de as mãos do operador estarem no caminho da lâmina.
2. *Detecção e remoção de perigos.* O sistema é projetado de modo que os perigos sejam detectados e removidos antes que resultem em um acidente. Por exemplo, um sistema de uma fábrica de produtos químicos pode detectar o excesso de pressão e abrir uma válvula de alívio para reduzir essas pressões antes que ocorra uma explosão.
3. *Limitação de danos.* O sistema pode incluir recursos de proteção que minimizem os danos que possam resultar em um acidente. Por exemplo, o motor de um avião inclui, normalmente, extintores de incêndio automáticos. Se ocorrer um incêndio, este poderá, em geral, ser controlado antes que represente uma ameaça para a aeronave.

Tabela 11.2 Terminologia de segurança

Termo	Definição
Acidente (ou desgraça)	Um evento não planejado ou uma sequência de eventos que resulta em morte ou dano pessoal, danos à propriedade ou ao meio ambiente. Uma *overdose* de insulina é um exemplo de acidente.
Perigo	Uma condição com o potencial de causar ou contribuir para um acidente. Uma falha do sensor que mede a glicose no sangue é um exemplo de um perigo.
Dano	Uma medida do prejuízo resultante de um acidente. Os danos podem variar desde muitas pessoas sendo mortas como resultado de um acidente a ferimentos leves ou danos materiais. Danos resultantes de uma overdose de insulina podem ser lesões graves ou a morte do usuário da bomba de insulina.
Severidade do perigo	Uma avaliação dos piores danos possíveis que poderiam resultar de um perigo. Severidade de perigo pode variar de catastrófico, em que muitas pessoas são mortas, a menor, em que os resultados são pequenos danos. Quando a morte de um indivíduo é uma possibilidade, uma avaliação razoável da severidade do perigo é 'muito elevado'.
Probabilidade de perigo	A probabilidade dos eventos que estão ocorrendo e são capazes de criar um perigo. Os valores de probabilidade tendem a ser arbitrários, mas variam de 'provável' (digamos 1/100 chance de ocorrência de perigo) a 'implausível' (sem situações concebíveis ou prováveis de ocorrência de perigo). A probabilidade de uma falha de sensor da bomba de insulina resultar em uma *overdose* provavelmente é baixa.
Risco	Essa é a medida da probabilidade de o sistema causar um acidente. O risco é avaliado considerando-se a probabilidade de perigo, a severidade do perigo e a probabilidade de o perigo causar um acidente. O risco de uma *overdose* de insulina é, provavelmente, médio a baixo.

Frequentemente, os acidentes acontecem quando várias coisas estão erradas ao mesmo tempo. Uma análise de acidentes graves (PERROW, 1984) sugere que quase todos aconteceram devido a uma combinação de falhas em partes diferentes de um sistema. Combinações inesperadas de falhas de subsistema levam a interações que resultaram em falha global de sistema. Por exemplo, a falha de um sistema de ar condicionado pode levar a um superaquecimento, o qual, em seguida, pode levar o hardware do sistema a gerar sinais incorretos. Perrow também sugere que não conseguimos prever todas as combinações possíveis de falhas. Os acidentes são, portanto, uma parte inevitável do uso de sistemas complexos.

Algumas pessoas têm usado isso como argumento contra o controle de software. Devido à complexidade do software, existem mais interações entre as diferentes partes de um sistema, o que significa que provavelmente haverá um número maior de combinações de defeitos que podem levar à falha de sistema.

No entanto, sistemas controlados por software podem monitorar uma gama maior de condições do que os sistemas eletromecânicos. Eles podem ser adaptados de forma relativamente fácil; usam hardware, cujo grau de confiabilidade inerente é elevado; e são fisicamente pequenos e leves. Os sistemas controlados por software podem fornecer intertravamentos sofisticados de segurança. Eles podem apoiar estratégias de controle que reduzem a quantidade de tempo necessário para as pessoas agirem em ambientes perigosos. Embora o controle de software possa introduzir mais maneiras como um sistema pode dar errado, ele também permite melhor monitorização e proteção e, portanto, pode contribuir para a melhoria da segurança de sistema.

Em todos os casos, é importante manter um senso de proporção sobre a segurança do sistema. É impossível fazer um sistema 100% seguro, e a sociedade precisa decidir se consequências de um acidente ocasional valem os benefícios do uso de tecnologias avançadas ou não. Também se trata de uma decisão política e social sobre como implantar recursos nacionais limitados para reduzir o risco para a população como um todo.

11.4 Proteção

A segurança é um atributo do sistema que reflete sua capacidade de se proteger de ataques externos, sejam acidentais ou deliberados. Esses ataques são possíveis porque a maioria dos computadores de uso geral está em rede e é, portanto, acessível a estranhos. Exemplos de ataques podem ser a instalação de vírus e cavalos de Troia, o uso não autorizado de serviços de sistema ou a modificação não autorizada de um sistema ou seus dados. Se você quer um sistema realmente seguro, é melhor não o conectar à Internet. Assim, seus problemas de proteção serão limitados a garantir que usuários autorizados não abusem do sistema. Na prática, porém, existem enormes benefícios no acesso à rede, não sendo rentável à maioria dos grandes sistemas desconectar-se da Internet.

Para alguns sistemas, a proteção é a dimensão mais importante da confiança de sistema. Sistemas militares, sistemas de comércio eletrônico e sistemas que envolvem processamento e intercâmbio de informações confidenciais, por exemplo, devem ser projetados de modo a alcançar um elevado nível de proteção. Se um sistema de reserva de passagens aéreas não estiver disponível, por exemplo, esse inconveniente pode causar alguns atrasos na emissão de bilhetes; ou, ainda, se o sistema não tiver proteção, o invasor pode, em seguida, apagar todas as reservas, tornando praticamente impossível às operações normais continuarem.

Como em outros aspectos de confiança, existe uma terminologia especializada associada à proteção. Alguns termos importantes, discutidos por Pfleeger (PFLEEGER e PFLEEGER, 2007), são definidos na Tabela 11.3. A Tabela 11.4 toma os conceitos de proteção descritos na Tabela 11.3 e mostra como eles se relacionam com o seguinte cenário do MHC-PMS:

> *O pessoal da clínica acessa o MHC-PMS com um nome de usuário e senha. O sistema requer que as senhas contenham pelo menos oito letras, mas permite que qualquer senha seja definida sem verificações adicionais. Um criminoso descobre que um astro do esporte está recebendo tratamento para problemas de saúde mental. Ele gostaria de obter acesso ilegal às informações desse sistema para poder chantagear o esportista.*
>
> *Fingindo ser um parente preocupado, ao falar com os enfermeiros na clínica de saúde mental ele descobre como acessar o sistema e as informações pessoais sobre os enfermeiros. Ao verificar os crachás, ele descobre os nomes de algumas pessoas autorizadas a acessarem o sistema. Ele, então, tenta fazer* logon *no sistema usando esses nomes tentando, sistematicamente, adivinhar senhas possíveis (como nomes dos filhos).*

Em qualquer sistema de rede, existem três principais tipos de ameaças à proteção:

1. *Ameaças à confidencialidade do sistema e seus dados.* Essas ameaças podem divulgar informações para pessoas ou programas não autorizados a acessarem-nas

Tabela 11.3 Terminologia de proteção

Termo	Definição
Ativo	Algo de valor que deve ser protegido. O ativo pode ser o próprio sistema de software ou dados usados por esse sistema.
Exposição	Possíveis perdas ou danos a um sistema de computação. Pode ser perda ou dano aos dados, ou uma perda de tempo e esforço, caso seja necessária a recuperação após uma brecha de proteção.
Vulnerabilidade	A fraqueza em um sistema computacional, que pode ser explorada para causar perdas ou danos.
Ataque	Uma exploração da vulnerabilidade de um sistema. Geralmente, vem de fora do sistema e é uma tentativa deliberada de causar algum dano.
Ameaças	Circunstâncias que têm potencial para causar perdas ou danos. Você pode pensar nisso como uma vulnerabilidade de um sistema submetido a um ataque.
Controle	Uma medida de proteção que reduz a vulnerabilidade do sistema. A criptografia é um exemplo de controle que reduz a vulnerabilidade de um sistema de controle de acesso fraco.

2. *Ameaças à integridade do sistema e seus dados*. Essas ameaças podem danificar o software ou corromper seus dados.
3. *Ameaças à disponibilidade do sistema e seus dados*. Essas ameaças podem restringir, para usuários autorizados, acesso ao software ou a seus dados.

Essas ameaças são, naturalmente, interdependentes. Se um ataque tornar o sistema indisponível, você não será capaz de atualizar as informações que mudam com tempo. Isso significa que a integridade do sistema pode estar comprometida. Se um ataque for bem-sucedido e a integridade do sistema for comprometida, então pode ser necessário parar para reparar o problema. Portanto, a disponibilidade do sistema ficará reduzida.

Na prática, a maioria das vulnerabilidades em sistemas sociotécnicos resulta de falhas humanas e não de problemas técnicos. As pessoas escolhem senhas fáceis de adivinhar ou anotam suas senhas em lugares onde podem ser encontradas. Os administradores de sistema cometem erros na configuração de controle de acesso ou arquivos de configuração, e os usuários não instalam ou não usam softwares de proteção. No entanto, como discutido na Seção 10.5, precisamos ter muito cuidado ao classificar o problema como um erro de usuário. Muitas vezes, os problemas humanos refletem decisões pobres, tomadas durante o projeto de sistema, por exemplo, a alteração frequente de senhas (que exige que os usuários anotem suas senhas) ou mecanismos de configurações complexos.

Os controles que você pode colocar em prática para melhorar a proteção de sistema são comparáveis àqueles de confiabilidade e segurança:

1. *Prevenção de vulnerabilidade*. Controles que se destinam a assegurar que os ataques não sejam bem-sucedidos. A estratégia aqui é o projeto do sistema para que os problemas da proteção sejam evitados. Por exemplo,

Tabela 11.4 Exemplos de terminologia de proteção

Termo	Exemplo
Ativo	Os registros de cada paciente que está recebendo ou recebeu tratamento.
Exposição	Potencial perda financeira de futuros pacientes que não procuram tratamento por não confiarem na clínica para manter seus dados. Prejuízo financeiro a partir de ação judicial pelo astro do esporte. Perda de reputação.
Vulnerabilidade	Um sistema de senhas fraco, que torna fácil para os usuários adivinharem as senhas. Senhas que são iguais aos nomes de usuários.
Ataque	Uma aparência de um usuário autorizado.
Ameaças	Um usuário não autorizado terá acesso ao sistema, adivinhando as credenciais (login e senha) de um usuário autorizado.
Controle	Um sistema de verificação de senhas que não permite senhas de usuários que sejam nomes próprios ou palavras que estão normalmente incluídas em um dicionário.

sistemas militares sensíveis não estão ligados às redes públicas, de modo a tornar impossível o acesso externo. Você deve pensar, também, na criptografia como um controle baseado na prevenção. Qualquer acesso não autorizado aos dados criptografados significa que estes não podem ser lidos pelo invasor. Na prática, é muito caro e demorado quebrar uma criptografia forte.

2. *Detecção e neutralização de ataques*. Controles que visam detectar e repelir os ataques. Esses controles incluem, em um sistema, funcionalidade que monitora sua operação e verifica padrões incomuns de atividade. Se tais padrões forem detectados, uma ação poderá ser tomada, como desligar partes do sistema, restringir o acesso a determinados usuários etc.
3. *Limitação de exposição e recuperação*. Controles que apoiam a recuperação de problemas. Podem variar desde estratégias de *backup* automatizadas e 'espelhamento' de informações para políticas de seguro que cubram os custos associados a um ataque bem-sucedido ao sistema.

Sem um nível razoável de proteção, não podemos estar confiantes quanto à disponibilidade, à confiabilidade e à segurança de um sistema. Métodos para a certificação de disponibilidade, confiabilidade e proteção assumem que um software operacional seja o mesmo software originalmente instalado. Se o sistema tiver sido atacado e o software tiver sido comprometido de alguma maneira (por exemplo, se o software foi modificado para incluir um *worm*), os argumentos de confiabilidade e de proteção não são mais válidos.

Erros no desenvolvimento de um sistema podem causar brechas de proteção. Se um sistema não responde às entradas inesperadas ou se limites de vetor não são verificados, os invasores podem, em seguida, explorar essas fraquezas para obter acesso ao sistema. Os principais incidentes de proteção, como o *worm* original de Internet (SPAFFORD, 1989) e o *worm* 'Code Red', aproveitam-se da mesma vulnerabilidade há mais de dez anos (BERGHEL, 2001). Programas em C# não incluem a verificação de limites de vetor, por isso é possível sobrescrever parte da memória com um código que permite o acesso não autorizado ao sistema.

- A falência de sistemas críticos de computação pode causar grandes prejuízos econômicos, perda de informação séria, danos físicos ou mesmo ameaças à vida humana.
- A confiança de um sistema computacional é uma propriedade que reflete o grau de confiança do usuário nesse sistema. As dimensões mais importantes da confiança são a disponibilidade, a confiabilidade, a segurança e a proteção.
- A disponibilidade de um sistema é a probabilidade de ele ser capaz de prestar serviços a seus usuários quando solicitado. A confiabilidade é a probabilidade de os serviços de sistema serem entregues conforme especificado.
- A confiabilidade está relacionada à probabilidade de ocorrência de um erro em uso operacional. Um programa pode conter defeitos conhecidos, mas ainda ser classificado como confiável pelos usuários, pois estes podem nunca usar recursos do sistema que sejam afetados pelos defeitos.
- A segurança de um sistema é um atributo que reflete a capacidade do sistema de funcionar, em condições normais ou não, sem causar danos a pessoas ou ao ambiente.
- A proteção reflete a capacidade de um sistema de se proteger contra ataques externos. Falhas de proteção podem levar a perda de disponibilidade, danos ao sistema ou aos dados, ou vazamento de informações para pessoas não autorizadas.
- Sem um nível razoável de proteção, a disponibilidade, a confiabilidade e a segurança do sistema podem ser comprometidas no caso de ataques externos causarem danos ao sistema. Se um sistema é não confiável, é difícil garantir sua proteção ou sua segurança, uma vez que esse sistema pode ser comprometido por falhas.

'The evolution of information assurance'. Um excelente artigo que discute a necessidade de proteger informações críticas de uma organização contra acidentes e ataques. (CUMMINGS, R. *IEEE Computer*, v. 35, n. 12, dez. 2002.) Disponível em: <http://dx.doi.org/10.1109/MC.2002.1106181>.

'Designing Safety Critical Computer Systems'. Essa é uma boa introdução à área de segurança dos sistemas críticos, que discute os conceitos fundamentais dos perigos e riscos. É mais acessível do que o livro de Dunn sobre

sistemas críticos de segurança. (DUNN, W.R. *IEEE Computer*, v. 36, n. 11, nov. 2003.) Disponível em: <http://dx.doi.org/10.1109/MC.2003.1244533>.

Secrets and Lies: Digital Security in a Networked World. Um excelente livro sobre proteção de computadores, muito lido, que aborda o assunto de uma perspectiva sociotécnica. As colunas de Schneier sobre questões de proteção (URL adiante) em geral também são muito boas. (SCHNEIER, B. *Secrets and Lies: Digital Security in a Networked World.* John Wiley & Sons, 2004.) Disponível em: <http://www.schneier.com/essays.html>.

EXERCÍCIOS

11.1 Sugira seis razões pelas quais, na maioria dos sistemas sociotécnicos, a confiança de software é importante.

11.2 Quais são as dimensões mais importantes da confiança de sistema?

11.3 Por que os custos de garantir a confiança aumentam exponencialmente à medida que os requisitos de confiabilidade aumentam?

11.4 Sugira quais atributos de confiança podem ser mais críticos para os seguintes sistemas. Justifique sua resposta.

- Um servidor de Internet fornecido por um ISP com milhares de clientes;
- Um bisturi controlado por computador, usado em cirurgias laparoscópicas;
- Um sistema de controle direcional, usado em um veículo lançador de satélites;
- Um sistema de gerenciamento de finanças pessoais baseado na Internet.

11.5 Identifique seis produtos de consumo que possam ser controlados por sistemas de software crítico de segurança.

11.6 Confiabilidde e segurança são atributos de confiança relacionados, porém distintos. Descreva as principais distinções entre esses atributos e explique por que é possível que um sistema confiável seja inseguro e vice-versa.

11.7 Em um sistema médico projetado para liberar radiação para tratamentos de tumores, sugira um possível perigo e proponha um recurso de software que possa ser usado para garantir que o perigo identificado não resulte em um acidente.

11.8 Em termos de proteção de computador, explique as diferenças entre um ataque e uma ameaça.

11.9 Usando o MHC-PMS como exemplo, identifique três ameaças a esse sistema (além da ameaça mostrada na Tabela 11.4). Baseado nessas ameaças, sugira controles que possam ser postos em prática para reduzir as chances de um ataque bem-sucedido.

11.10 Você é um especialista em proteção de computadores e foi abordado por uma organização que luta pelos direitos das vítimas de tortura. Você foi convidado a ajudar a organização a ter acesso não autorizado aos sistemas informáticos de uma empresa norte-americana. Isso os ajudará a confirmar ou negar que essa empresa está vendendo equipamentos usados na tortura de presos políticos. Discuta os dilemas éticos que essa solicitação levanta e como você reagiria a ela.

REFERÊNCIAS

ADAMS, E. N. Optimizing preventative service of software products. *IBM J. Res & Dev.*, v. 28, n. 1, 1984, p. 2-14.

BERGHEL, H. The Code Red Worm. *Comm. ACM*, v. 44, n. 12, 2001, p. 15-19.

BOEHM, B. W.; McCLEAN, R. L.; URFIG, D. B. Some experience with automated aids to the design of large-scale reliable software. *IEEE Trans. on Software Engineering*, v. SE-1, n. 1, 1975, p. 125-133.

ELLISON, R.; LINGER, R.; LIPSON, H.; MEAD, N.; MOORE, A. Foundations of Survivable Systems Engineering. *Crosstalk: The Journal of Defense Software Engineering*, v. 12, 2002, p. 10-15.

ELLISON, R. J.; FISHER, D. A.; LINGER, R. C.; LIPSON, H. F.; LONGSTAFF, T. A.; MEAD, N. R. Survivability: Protecting Your Critical Systems. *IEEE Internet Computing*, v. 3, n. 6, 1999a, p. 55-63.

ELLISON, R. J.; LINGER, R. C.; LONGSTAFF, T.; MEAD, N. R. Survivable Network System Analysis: A Case Study. *IEEE Software*, v. 16, n. 4, 1999b, p. 70-77.

ENDRES, A. An analysis of errors and their causes in system programs. *IEEE Trans. on Software Engineering*, v. SE-1, n. 2, 1975, p. 140-149.

LAPRIE, J.-C. Dependable Computing: Concepts, Limits, Challenges. *FTCS-25*: 25th IEEE Symposium on Fault-Tolerant Computing, Pasadena, Calif.: IEEE Press, 1995.

LITTLEWOOD, B. Software Reliability Growth Models. In: *Software Reliability Handbook*. ROOK, P. (Org.). Amsterdam: Elsevier, 1990, p. 401-412.

LUTZ, R. R. Analysing Software Requirements Errors in Safety-Critical Embedded Systems. *RE'93*, San Diego, Calif: IEEE 1993.

MILLS, H. D.; DYER, M.; LINGER, R. Cleanroom Software Engineering. *IEEE Software*, v. 4, n. 5, 1987, p. 19-25.

NAKAJO, T.; KUME, H. A Case History Analysis of Software Error-Cause Relationships. *IEEE Trans. on Software Eng*, v. 18, n. 8, 1991, p. 830-838.

PERROW, C. *Normal Accidents*: Living with High-Risk Technology. Nova York: Basic Books, 1984.

PFLEEGER, C. P.; PFLEEGER, S. L. *Security in Computing*. 4. ed. Boston: Addison-Wesley, 2007.

SPAFFORD, E. The Internet Worm: Crisis e Aftermath. *Comm. ACM*, v. 32, n. 6, 1989, p. 678-687.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 **12** 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Especificação de confiança e proteção

Objetivos

O objetivo deste capítulo é explicar como especificar requisitos funcionais e não funcionais de confiança e de proteção. Depois de ler este capítulo, você:

- entenderá como uma abordagem dirigida a riscos pode ser usada para identificação e análise de requisitos de segurança, de confiabilidade e de proteção;
- entenderá como as árvores de defeitos podem ajudar a analisar os riscos e derivar requisitos de segurança;
- conhecerá as métricas para especificação de confiabilidade e como elas são usadas para especificar requisitos mensuráveis de confiabilidade;
- conhecerá os diferentes tipos de requisitos de proteção que podem ser necessários em um sistema complexo;
- estará ciente das vantagens e desvantagens de usar especificações matemáticas formais de um sistema.

Conteúdo

Em setembro de 1993, um avião aterrissou no aeroporto de Varsóvia, na Polônia, durante uma tempestade. Por nove segundos após a aterrissagem, os freios do sistema, controlado por computador, não funcionaram. O sistema de frenagem não reconheceu que o avião havia aterrissado e assumiu que a aeronave ainda estava no ar. Um recurso de segurança da aeronave havia parado a entrada do sistema reverso de empuxo, que diminui a velocidade da aeronave, pois se o avião estiver no ar isso pode ser perigoso. O avião passou do limite final da pista, atingiu um banco de terra e pegou fogo.

O inquérito sobre o acidente revelou que o software do sistema de frenagem tinha operado de acordo com sua especificação. Não houve erros no programa. No entanto, a especificação de software foi incompleta e não levou em consideração uma situação rara, que surgiu no caso apresentado. O software funcionou, mas o sistema falhou.

Esse caso mostra que a confiança de sistema não depende apenas da boa engenharia, mas também exige atenção aos detalhes quando os requisitos de sistema são derivados e ocorre a inclusão de requisitos especiais de software orientados a garantir a confiança e a proteção de um sistema. Esses requisitos de confiança e proteção são de dois tipos:

1. Requisitos funcionais, que definem a verificação e os recursos de recuperação a serem incluídos no sistema e os recursos que fornecem proteção contra falhas de sistema e ataques externos.
2. Requisitos não funcionais, que definem a confiabilidade e a disponibilidade requeridas do sistema.

Muitas vezes, o ponto de partida para gerar requisitos funcionais de confiança e proteção são regras, políticas e regulamentos de negócio ou de domínio em alto nível. Esses são requisitos de alto nível que talvez sejam mais bem-descritos como requisitos do tipo 'não deve'. Em contrapartida, assim como os requisitos funcionais normais definem o que o sistema deve fazer, os requisitos 'não deve' definem comportamentos inaceitáveis do sistema. Exemplos de requisitos do tipo 'não deve' são:

'O sistema não deve permitir que usuários modifiquem permissões de acesso em arquivos que eles não criaram.' (proteção)

'O sistema não deve permitir o modo reverso de empuxo quando a aeronave estiver em voo.' (segurança)

'O sistema não deve permitir a ativação simultânea de mais de três sinais de alarme.' (segurança)

Esses requisitos não podem ser implementados diretamente, mas precisam ser decompostos em requisitos funcionais de software mais específicos. Como alternativa, eles podem ser implementados por meio de decisões de projeto de sistema, como uma decisão de usar determinados tipos de equipamentos no sistema.

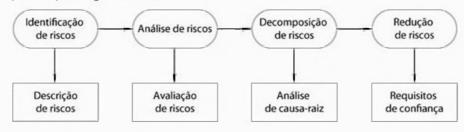
12.1 Especificação de requisitos dirigida a riscos

Os requisitos de confiança e proteção podem ser pensados como requisitos de proteção como um todo. Eles especificam como um sistema deve se proteger de defeitos internos, parar falhas de sistema que causam danos ao meio ambiente, parar acidentes ou ataques do ambiente do sistema que estejam danificando o próprio sistema, bem como facilitar a recuperação em caso de falha. Para descobrir esses requisitos de proteção, você precisa entender os riscos para o sistema e seu ambiente. Uma abordagem dirigida a riscos para a especificação de requisitos leva em consideração os eventos perigosos que podem ocorrer, a probabilidade de que estes eventos venham a ocorrer, a probabilidade de os resultados desses eventos serem danos e a extensão dos danos causados. Os requisitos de proteção e confiança podem ser estabelecidos com base na análise das possíveis causas de eventos perigosos.

A especificação dirigida a riscos é uma abordagem amplamente usada por desenvolvedores de sistemas de segurança e proteção críticos. Ela incide sobre eventos que possam causar danos maiores ou que sejam suscetíveis de ocorrer frequentemente. Eventos que tenham consequências pequenas ou que sejam extremamente raros podem ser ignorados. Em sistemas críticos de segurança, os riscos estão associados a perigos que possam resultar em acidentes; em sistemas críticos de proteção, os riscos são provenientes de ataques internos ou externos em um sistema, os quais se destinam a explorar possíveis vulnerabilidades.

Um processo geral de especificação dirigida a riscos (Figura 12.1) envolve a compreensão dos riscos enfrentados pelo sistema, descobrir suas causas e gerar condições para gerenciar esses riscos. Os estágios desse processo são:

1. *Identificação de riscos.* Potenciais riscos para o sistema são identificados. Eles dependem do ambiente em que o sistema será usado. Os riscos podem surgir a partir das interações entre o sistema e as condições especiais de seu ambiente operacional. O acidente de Varsóvia, discutido anteriormente, aconteceu quando os ventos durante uma tempestade causaram a inclinação do avião, de forma que ele pousou em uma única roda, e não duas (o que raramente acontece).
2. *Análise e classificação de riscos.* Cada risco é considerado individualmente. Riscos potencialmente graves e plausíveis são selecionados para análises posteriores. Nesse estágio, os riscos podem ser eliminados, porque não são suscetíveis de surgir ou porque não podem ser detectados pelo software (por exemplo, uma reação alérgica ao sensor do sistema de bomba de insulina).

Figura 12.1 Especificação dirigida a riscos

3. *Decomposição de riscos*. Cada risco é analisado para descobrir suas causas-raízes potenciais. As causas-raízes são as razões pelas quais um sistema pode falhar. Estas podem ser erros de software, de hardware, ou, ainda, vulnerabilidades inerentes resultantes de decisões de projeto de sistema.
4. *Redução de riscos*. Existem propostas para que os riscos identificados possam ser reduzidos ou eliminados. Elas contribuem para os requisitos de confiança de sistema, os quais definem as defesas contra os riscos e como os riscos serão gerenciados.

Para grandes sistemas, a análise de risco pode ser estruturada em fases (LEVESON, 1995), em que cada fase considera diferentes tipos de riscos:

1. Análise preliminar de risco, em que os principais riscos do ambiente do sistema são identificados. Estes independem da tecnologia usada para o desenvolvimento de sistema. O objetivo da análise preliminar de risco é o desenvolvimento de um conjunto inicial de requisitos de proteção e de confiança para o sistema.
2. Análise de risco de ciclo de vida, que ocorre durante o desenvolvimento do sistema e se preocupa com os riscos decorrentes das decisões de projeto de sistema. Diferentes tecnologias e arquiteturas de sistema têm seus próprios riscos associados. Nesse estágio, você deve ampliar os requisitos para se proteger contra esses riscos.
3. Análise de risco operacional, que se preocupa com a interface de usuário do sistema e os riscos resultantes dos erros de operador. Novamente, uma vez que as decisões foram tomadas no projeto de interface, outros requisitos de proteção podem precisar ser adicionados.

Essas fases são necessárias, pois é impossível tomar todas as decisões de confiança e proteção sem informações completas sobre a implementação de sistema. Os requisitos de proteção e confiança são particularmente afetados por escolhas de tecnologia e decisões de projeto. Verificações de sistema podem ser incluídas para garantir que os componentes de terceiros funcionem corretamente. Os requisitos de proteção podem precisar ser modificados porque entram em conflito com os recursos de proteção fornecidos por um sistema de prateleira.

Por exemplo, um requisito de segurança pode ser um em que, em vez de uma senha, os usuários devem identificar-se a um sistema com uma sequência de palavras. As sequências de palavras são consideradas mais seguras do que senhas. Elas são mais difíceis de serem adivinhadas por um intruso ou de serem descobertas por meio do uso de um sistema de quebra de senha automatizada. No entanto, se for tomada a decisão de usar um sistema existente que suporte apenas autenticação baseada em senha, então esse requisito de proteção não pode ser suportado. Assim, pode ser necessário incluir funcionalidade adicional ao sistema para compensar o aumento de riscos de se usar senhas, e não sequências de palavras.

12.2 Especificação de segurança

Sistemas de segurança críticos são aqueles nos quais as falhas podem afetar o ambiente do sistema e causar ferimentos ou morte a pessoas nesse ambiente. A principal preocupação é a especificação de segurança para identificar os requisitos que minimizarão a probabilidade de ocorrência de falhas no sistema. Os requisitos de segurança são primordialmente requisitos de proteção, e não estão preocupados com o funcionamento normal do sistema. Eles podem definir que o sistema deve ser desligado para manter a segurança. Portanto, ao derivar os requisitos de segurança, você precisa encontrar um equilíbrio aceitável entre a segurança e a funcionalidade e evitar a superproteção. Não há sentido em construir um sistema muito seguro se ele não funciona de forma eficaz.

Lembre-se da discussão no Capítulo 10, os sistemas de segurança críticos usam uma terminologia especializada, na qual um perigo é algo que poderia (embora não necessariamente) resultar em morte ou lesão de uma pessoa, e um risco é a probabilidade de o sistema entrar em um estado perigoso. Portanto, a especificação de segurança geralmente se centra nos perigos que podem surgir em determinada situação e nos eventos que podem causar esses perigos.

As atividades do processo geral de especificação baseada em riscos, mostradas na Figura 12.1, organizam-se para o processo de especificação de segurança da seguinte maneira:

1. *Identificação de riscos*. Em especificação de segurança, esse é o processo de identificação de perigos que identificam os riscos que podem ameaçar o sistema.
2. *Análise de riscos*. Esse é o processo de avaliação de riscos para decidir quais situações são mais perigosas e/ou mais prováveis. Estas devem ser priorizadas ao derivar os requisitos de segurança.

3. *Decomposição de riscos*. Esse processo pretende descobrir os eventos que podem ocasionar um perigo. Em especificação de segurança, o processo é conhecido como análise de riscos.
4. *Redução de riscos*. Esse processo é baseado no resultado da análise de perigos e conduz à identificação de requisitos de segurança. Estes podem estar preocupados em garantir que perigos não surjam ou que não conduzam a um acidente ou, ainda, se ocorrer um acidente, que os danos associados sejam minimizados.

12.2.1 Identificação de perigos

Em sistemas de segurança críticos, os principais riscos provêm de perigos que podem levar a um acidente. Você pode resolver o problema de identificação de perigos considerando diferentes tipos de riscos, como perigos físicos, perigos elétricos, perigos biológicos, perigos de radiação, perigos de falha de serviço, e assim por diante. Cada uma dessas classes pode ser analisada para se descobrir perigos específicos que possam ocorrer. Possíveis combinações de perigos potencialmente danosas também devem ser identificadas.

O sistema de bomba de insulina que usei como exemplo nos capítulos anteriores é um sistema crítico de segurança, pois uma falha pode causar lesões ou até mesmo a morte ao usuário do sistema. Os acidentes que podem ocorrer ao se usar essa máquina incluem o usuário sofrer, a longo prazo, as consequências do controle ruim das taxas de açúcar no sangue (problemas nos olhos, coração e rins), além de disfunção cognitiva, como resultado de baixos níveis de açúcar no sangue, ou a ocorrência de alguma outra condição médica, tal qual uma reação alérgica.

Alguns dos perigos do sistema de bomba de insulina são:

- cálculo de overdose de insulina (falha de serviço);
- cálculo de subdosagem de insulina (falha de serviço);
- falha do sistema de monitoramento de hardware (falha de serviço);
- falha de energia devido a bateria esgotada (elétrico);
- interferência elétrica com outros equipamentos médicos, como um marcapasso cardíaco (elétrico);
- mau contato de sensor e atuador, causado por instalação incorreta (físico);
- quebra de partes da máquina no corpo do paciente (físico);
- infecção causada pela introdução da máquina (biológico);
- reação alérgica aos materiais ou à insulina usada na máquina (biológico).

Engenheiros experientes, trabalhando com especialistas e consultores profissionais de segurança, identificam perigos nas experiências anteriores e a partir de uma análise de domínio da aplicação. Técnicas de trabalho em grupo, como '*brainstorming*', podem ser usadas. Para o sistema de bomba de insulina, as pessoas envolvidas podem incluir médicos, cientistas médicos, engenheiros e projetistas de software.

Geralmente, os perigos relacionados ao software estão relacionados com falhas na entrega de um serviço de sistema ou com a falha de sistemas de monitoramento e proteção. Os sistemas de monitoramento e proteção são incluídos em um dispositivo para detectar condições, como níveis de bateria fraca, que possam levar à falha de dispositivo.

12.2.2 Avaliação de perigos

O processo de avaliação de perigos concentra-se em entender a probabilidade de ocorrer um perigo e as consequências, em caso de um acidente ou incidente associado à ocorrência desse perigo. Você precisa fazer essa análise para entender se um perigo é uma séria ameaça ao sistema ou ambiente. A análise também lhe fornece uma base para decidir sobre a forma de gerenciar o risco associado ao perigo.

Para cada perigo, o resultado do processo de análise e classificação é uma declaração de aceitabilidade. Isso é expresso em termos de risco, em que o risco leva em conta a probabilidade de um acidente e suas consequências. Existem três categorias de risco que podem ser usadas na avaliação de risco:

1. Riscos intoleráveis, em sistemas de segurança críticos — são aqueles que ameaçam a vida humana. O sistema deve ser projetado de modo que esses riscos não possam surgir ou, se surgirem, que recursos do sistema garantam sua detecção antes que provoquem um acidente. No caso da bomba de insulina, um risco intolerável é uma overdose de insulina.

2. Riscos tão baixos quanto razoavelmente práticos (ALARP, do inglês *as low as reasonably practical*) são aqueles cujas consequências são menos graves ou graves, mas têm uma probabilidade muito baixa de ocorrência. O sistema deve ser projetado de modo que a probabilidade de um acidente decorrer de um perigo seja minimizada e sujeita a outras considerações, como custo e entrega. Um risco ALARP para uma bomba de insulina pode ser a falha do sistema de monitoramento de hardware. As consequências são, na pior das hipóteses, uma subdosagem de insulina de curta duração. Essa é uma situação que não provocaria um acidente grave.
3. Riscos aceitáveis são aqueles em que, geralmente, os acidentes associados resultam em danos menores. Projetistas de sistema devem tomar todas as medidas possíveis para reduzir os riscos 'aceitáveis', desde que isso não aumente os custos, os prazos de entrega ou outros atributos não funcionais de sistema. Um risco aceitável no caso da bomba de insulina pode ser o risco de uma reação alérgica surgir no usuário; normalmente, isso provoca apenas irritação de pele. Não valeria a pena usar materiais especiais, mais caros, no dispositivo, para reduzir esse risco.

A Figura 12.2 (BRAZENDALE e BELL, 1994), desenvolvida para sistemas de segurança críticos, mostra as três regiões. A forma do diagrama reflete os custos de assegurar que os riscos não resultem em incidentes ou acidentes. O custo do projeto de sistema para lidar com o risco é indicado pela largura do triângulo. Os maiores custos são incorridos por riscos na parte superior do diagrama, e os custos mais baixos, por riscos no vértice do triângulo.

As fronteiras entre as regiões na Figura 12.2 não são técnicas, mas dependem de fatores sociais e políticos. Ao longo do tempo, a sociedade tornou-se mais avessa aos riscos, assim, as fronteiras se mudaram para baixo. Embora os custos financeiros da aceitação de riscos e de pagamento por quaisquer acidentes resultantes possam ser inferiores aos custos de prevenção de acidentes, a opinião pública pode exigir que o dinheiro seja investido na redução da probabilidade de um acidente de sistema, o que geraria custos adicionais.

Por exemplo, pode ser mais barato para uma empresa limpar a poluição nas raras ocasiões em que ocorrer do que investir na instalação de sistemas de prevenção da poluição. No entanto, como o público e a imprensa não vão tolerar esses acidentes, não é mais aceitável limpar os danos em vez de prevenir o acidente. Esses eventos também podem causar uma reclassificação do risco. Desse modo, os riscos que foram pensados para serem improváveis (e, consequentemente, na região ALARP) podem ser reclassificados como intoleráveis em virtude de eventos como ataques terroristas ou de outros acidentes que tenham ocorrido.

A avaliação de perigos envolve estimar a probabilidade de perigo e gravidade de risco, o que geralmente é difícil, pois os riscos e os acidentes são raros, e os engenheiros envolvidos podem não ter experiência direta com incidentes ou acidentes anteriores. Probabilidades e severidades são atribuídas por meio de termos relativos como 'provável', 'pouco provável', 'raro' e 'alto', 'médio' e 'baixo'. Só é possível quantificar esses termos caso estejam disponíveis dados de incidentes suficientes para a análise estatística.

A Tabela 12.1 apresenta uma classificação de risco para os riscos identificados na seção anterior para o sistema de administração de insulina. Eu separei os perigos que se relacionam com o cálculo incorreto de insulina no caso de uma overdose e de uma dose insuficiente de insulina. No curto prazo, uma overdose de insulina é potencialmente mais grave do que uma dose insuficiente de insulina. A overdose de insulina pode resultar em disfunção

Figura 12.2 O triângulo de risco

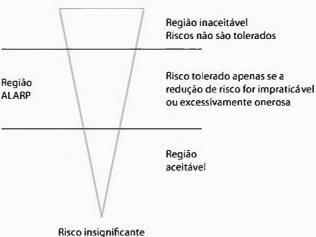



Tabela 12.1 Classificação de riscos para a bomba de insulina

Perigo identificado	Probabilidade de perigo	Severidade de acidente	Risco estimado	Aceitabilidade
1. Cálculo de overdose de insulina	Média	Alta	Alto	Intolerável
2. Cálculo de dose insuficiente de insulina	Média	Baixa	Baixo	Aceitável
3. Falha de sistema de monitoramento de hardware	Média	Média	Baixo	ALARP
4. Falha de energia	Alta	Baixa	Baixo	Aceitável
5. Máquina ajustada incorretamente	Alta	Alta	Alto	Intolerável
6. Quebra de máquina no paciente	Baixa	Alta	Médio	ALARP
7. Máquina causa infecção	Média	Média	Médio	ALARP
8. Interferência elétrica	Baixa	Alta	Médio	ALARP
9. Reação alérgica	Baixa	Baixa	Baixo	Aceitável

cognitiva, coma e morte. Dose insuficiente de insulina leva a altos níveis de açúcar no sangue. No curto prazo, isso causa cansaço, não muito grave; mas no longo prazo, pode levar a problemas sérios de coração, rins e olhos.

Os perigos 4 a 9 na Tabela 12.1 não estão relacionados ao software, mas ele, no entanto, tem um papel a desempenhar na detecção de perigos. O software de monitoramento de hardware deve acompanhar o estado de sistema e avisar possíveis problemas. Normalmente, a advertência permitirá a detecção do perigo antes que este cause um acidente. Exemplos de perigos que podem ser detectados são a falha de energia, detectada pelo monitoramento da bateria, e o posicionamento incorreto da máquina, detectado pelo monitoramento de sinais do sensor de açúcar no sangue.

Certamente, o software de monitoramento de sistema é relacionado com a segurança. Falhas na detecção de perigos podem resultar em acidentes. Se o sistema de monitoramento falhar, mas o hardware continuar funcionando corretamente, então essa não é uma falha grave. No entanto, se o sistema de monitoramento falhar, e uma falha no hardware não puder ser detectada, então consequências mais sérias podem surgir.

12.2.3 Análise de perigos

A análise de perigos é o processo de descobrir as causas-raízes dos perigos em um sistema crítico de segurança. Seu objetivo é descobrir quais eventos ou combinações de eventos podem causar uma falha no sistema que resulte em um perigo. Para fazer isso, você pode usar uma abordagem '*top-down*' ou '*bottom-up*'. As técnicas *top-down* são dedutivas e tendem a ser mais fáceis de usar; começam com o perigo e, a partir disso, trabalham as possíveis falhas de sistema. As técnicas *bottom-up* são indutivas, começam com uma falha de sistema proposta e identificam os perigos que poderiam resultar dela.

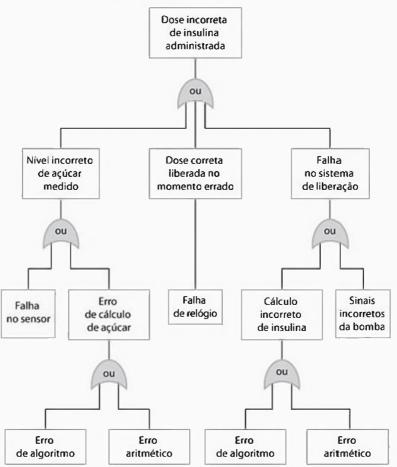
Várias técnicas têm sido propostas como possíveis abordagens para a decomposição ou análise de perigos. Elas são resumidas por Storey (1996). Incluem revisões e *checklists*, técnicas formais, como a análise de rede de Petri (PETERSON, 1981), a lógica formal (JAHANIAN e MOK, 1986) e análises de árvore de defeitos (LEVESON e STOLZY, 1987; STOREY, 1996). Como não tenho espaço nesta obra para abordar todas essas técnicas, concentro-me em uma abordagem amplamente usada para análise de risco, baseada em árvores de defeitos. Essa técnica é bastante fácil de compreender, mesmo sem domínio de conhecimento especializado.

Uma análise da árvore de defeitos inicia-se com os perigos identificados. Para cada perigo, você volta atrás para descobrir suas possíveis causas. Você coloca o perigo na raiz da árvore e identifica os estados de sistema que podem levar a esse perigo. Depois, identifica os estados de sistema que levam a cada um desses estados. E continua essa decomposição, até chegar à(s) causa(s)-raiz(es) do risco. Geralmente, os perigos que surgem de uma única combinação de causas-raízes são menos suscetíveis de gerar um acidente do que os perigos com uma causa-raiz única.

A Figura 12.3 é uma árvore de defeitos para os perigos relacionados com o software do sistema de administração da insulina que pode causar a entrega de uma dose incorreta. Nesse caso, mesclei uma dose insuficiente e uma overdose de insulina em um único perigo, ou seja, 'dose de insulina administrada incorretamente'. Isso reduz o número necessário de árvores de defeitos. Naturalmente, quando você especifica como o software deve reagir a esse perigo, é preciso distinguir entre uma dose insuficiente e uma overdose de insulina. Como dito anteriormente, esses perigos não são igualmente graves; a curto prazo, uma overdose é o perigo mais grave.

A partir da Figura 12.3, você pode ver que:

1. Existem três condições capazes de conduzir a administração de uma dose incorreta de insulina. O nível de açúcar no sangue pode ter sido incorretamente medido, tal que o requisito de insulina foi calculado com uma entrada incorreta. O sistema de liberação pode não responder corretamente aos comandos, especificando a quantidade de insulina a ser injetada. Outra possibilidade seria a dose ser corretamente calculada, mas ser entregue muito cedo ou muito tarde.
2. O ramo esquerdo da árvore de defeitos, relacionado com a medida errada do nível de açúcar no sangue, indica como isso poderia acontecer. Isso pode ocorrer porque o sensor que calcula o nível de açúcar falhou ou porque o cálculo do nível de açúcar no sangue foi realizado incorretamente. O nível de açúcar é calculado a partir de algum parâmetro medido, tal como a condutividade da pele. O cálculo incorreto pode resultar tanto em um algoritmo incorreto como em um erro aritmético que resulta do uso de números de ponto flutuante.
3. O ramo central da árvore está relacionado com os problemas de *timing* e conclui que estes só podem resultar de falhas no relógio do sistema.
4. O ramo direito da árvore, relacionado com a falha do sistema de liberação, analisa as possíveis causas da falha. Estas podem resultar de um cálculo incorreto do requisito de insulina ou de uma falha no envio de sinais corre-

Figura 12.3 Um exemplo de uma árvore de defeitos

tos para a bomba que libera a insulina. Mais uma vez, um cálculo incorreto pode ser resultado de erros de falha de algoritmo ou de erros aritméticos.

As árvores de defeitos também são usadas para identificar potenciais problemas de hardware. As árvores de defeitos de hardware podem fornecer indicações para requisitos de software para detectar e, eventualmente, corrigir esses problemas. Por exemplo, as doses de insulina não são administradas em uma frequência muito alta — não mais que duas ou três vezes por hora, e, às vezes, com menos frequência do que isso. Portanto, a capacidade do processador está disponível para executar programas de diagnóstico e de autoverificação. Os erros de hardware, como erros de sensor, bomba e relógio etc. podem ser descobertos e advertidos antes de terem consequências sérias para o paciente.

12.2.4 Redução de riscos

Uma vez que os riscos potenciais e suas causas-raízes tenham sido identificados, você é capaz de derivar os requisitos de segurança que gerenciam os riscos e garantir que incidentes ou acidentes não ocorram. Existem três possíveis estratégias que você pode usar:

1. *Prevenção de perigos*. O sistema é projetado para que o perigo não possa ocorrer.
2. *Detecção e remoção de perigos*. O sistema é projetado de modo que os perigos sejam detectados e neutralizados antes que resultem em um acidente.
3. *Limitação de danos*. O sistema é projetado de modo que as consequências de um acidente sejam minimizadas.

Em geral, os projetistas de sistemas críticos usam uma combinação dessas abordagens. Em um sistema de segurança crítico, perigos intoleráveis podem ser manipulados para minimizar sua probabilidade e adicionar um sistema de proteção que ofereça um *backup* de segurança. Por exemplo, em um sistema de controle de planta química, o sistema tentará detectar e evitar excesso de pressão no reator. No entanto, pode haver também um sistema de proteção independente que monitore a pressão e abra uma válvula de alívio caso seja detectada pressão alta.

No sistema de liberação de insulina, um 'estado de segurança' é um estado de desligamento em que a insulina não é injetada. Durante um curto período, isso não é uma ameaça para a saúde do diabético. Para as falhas de software que poderiam levar a uma dose incorreta de insulina, as seguintes 'soluções' podem ser aplicadas:

1. *Erro aritmético*. Isso pode ocorrer quando um cálculo aritmético causa uma falha de representação. A especificação deve identificar todos os erros aritméticos que podem ocorrer e estabelecer que um tratador de exceção deva ser incluído para cada erro possível. A especificação deve definir as ações a serem tomadas para cada um desses erros. A ação-padrão de segurança é desligar o sistema de entrega e ativar um alarme de aviso.
2. *Erro de algoritmo*. Essa é uma situação mais difícil, pois não existe um programa de exceção claro a ser tratado. Esse tipo de erro pode ser detectado por meio da comparação da dose de insulina necessária com a dose anteriormente liberada. Se essa for muito maior, pode significar que a quantidade foi calculada incorretamente. O sistema também pode acompanhar a sequência da dose; depois de liberar algumas doses acima da média, um aviso pode ser emitido, e a dosagem posterior, limitada.

Alguns dos requisitos de segurança resultantes para o software de bomba de insulina são mostrados no Quadro 12.1. Tratam-se dos requisitos de usuário e, naturalmente, seriam expressos em mais detalhes na especificação de requisitos de sistema. No Quadro 12.1, as referências às tabelas 3 e 4 estão relacionadas às tabelas incluídas nos documentos de requisitos, não mostrados aqui.

Quadro 12.1 Exemplos de requisitos de segurança (RS, do inglês *safety requirements*)

RS1: O sistema não deve liberar uma única dose de insulina maior que a dose máxima especificada para um usuário do sistema.
RS2: O sistema não deve liberar uma dose diária acumulada de insulina maior que uma dose diária máxima especificada para um usuário do sistema.
RS3: O sistema deve incluir um recurso de diagnóstico de hardware que deve ser executado pelo menos quatro vezes por hora.
RS4: O sistema deve incluir um tratador de exceção para todas as exceções que são identificadas na Tabela 3.
RS5: O alarme acústico deve ser soado quando for descoberta qualquer anomalia de hardware ou software, e uma mensagem de diagnóstico, tal como definido na Tabela 4, deve ser exibida.
RS6: Em caso de alarme, a liberação de insulina deve ser suspensa até que o usuário reinicie o sistema e limpe o alarme.

12.3 Especificação de confiabilidade

Como discutido no Capítulo 10, a confiabilidade geral de um sistema depende da confiabilidade do hardware, da confiabilidade do software e da confiabilidade dos operadores do sistema. O software de sistema deve levar isso em conta, além de incluir os requisitos que compensam as falhas de software. Também pode haver requisitos de confiabilidade relacionados a ajudar a detectar e recuperar falhas de hardware e erros de operador.

A confiabilidade é diferente da segurança e da proteção, no sentido de que é um atributo mensurável do sistema. Ou seja, é possível especificar o nível de confiabilidade necessário, acompanhar a operação do sistema ao longo do tempo e verificar se a confiabilidade necessária foi alcançada. Por exemplo, um requisito de confiabilidade pode ser um em que as falhas de sistema que necessitam de reiniciação não devem ocorrer mais de uma vez por semana. Cada vez que tal falha ocorre, ela pode ser registrada e você pode verificar se o nível de confiabilidade foi alcançado. Se não, você pode alterar seu requisito de confiabilidade ou submeter um pedido de alteração para resolver os problemas subjacentes ao sistema. Você pode decidir aceitar um nível de confiabilidade menor, devido aos custos de mudança do sistema para melhorar a confiabilidade ou porque a correção do problema pode ter efeitos colaterais adversos, como baixo desempenho ou rendimento inferior ao esperado.

Em contrapartida, a segurança e a proteção são relacionadas a como evitar situações indesejáveis, em vez de especificar um 'nível' desejado de segurança ou proteção. Mesmo uma única situação dessas, em todo o período de vida de um sistema, pode ser inaceitável, e, caso ocorra, mudanças no sistema precisarão ser feitas. Não faz sentido fazer declarações como 'defeitos de sistema devem resultar em menos de dez danos por ano'. Assim que um dano ocorre, o problema do sistema deve ser corrigido.

Os requisitos de confiabilidade são, portanto, de dois tipos:

1. Requisitos não funcionais, que definem o número de falhas aceitáveis durante o uso normal do sistema, ou o tempo em que o sistema não está disponível para uso. Esses são os requisitos de confiabilidade quantitativa.
2. Requisitos funcionais, que definem as funções de sistema e de software que evitam, detectam ou toleram defeitos no software e, assim, garantem que esses defeitos não gerem à falha de sistema.

Os requisitos quantitativos de confiabilidade conduzem aos requisitos funcionais de sistema relacionados. Para atingir um nível requerido de confiabilidade, os requisitos funcionais e de projeto devem especificar os defeitos a serem detectados e as ações que devem ser tomadas para garantir que esses defeitos não causem falhas de sistema.

O processo de especificação de confiabilidade pode ser baseado no processo de especificação geral dirigido a riscos, mostrado na Figura 12.1:

1. *Identificação de riscos*. Nessa etapa, você identifica os tipos de falhas de sistema que podem levar a perdas econômicas de algum tipo. Por exemplo, um sistema de comércio eletrônico pode estar indisponível, de modo que os clientes não consigam fazer encomendas, ou uma falha que corrompe os dados pode exigir tempo para que o banco de dados seja restaurado a partir de um *backup* e para que as transações previamente processadas sejam executadas novamente. A lista de tipos possíveis de falha, mostrada na Tabela 12.2, pode ser o ponto de partida para a identificação de riscos.

Tabela 12.2 Tipos de falha de sistema

Tipo de falha	Descrição
Perda de serviço	O sistema está indisponível e não pode oferecer seus serviços aos usuários. Pode ser dividida em perda de serviços críticos e perda de serviços não críticos, em que as consequências de uma falha em serviços não críticos são menores do que as de falha em serviço crítico.
Entrega incorreta de serviço	O sistema não oferece os serviços de forma correta. Novamente, pode ser especificada em termos de erros menores e maiores ou erros na entrega de serviços críticos e não críticos.
Corrupção de sistema/dados	A falha do sistema provoca danos ao próprio sistema ou a seus dados. Geralmente, acontece em conjunto com outros tipos de falhas, embora não necessariamente seja dessa forma.

2. *Análise de riscos.* Envolve estimar os custos e as consequências dos diferentes tipos de falhas de software e selecionar as falhas com grandes consequências, para análise posterior.
3. *Decomposição de riscos.* Nessa fase, você faz uma análise de causa-raiz de falhas sérias e possíveis de sistema. No entanto, isso pode ser impossível na fase de requisitos, pois as causas podem depender de decisões do projeto de sistema. Você pode ter de voltar a essa atividade durante o projeto e o desenvolvimento.
4. *Redução de riscos.* Nessa etapa, você deve gerar especificações quantitativas de confiabilidade que estabeleçam as probabilidades aceitáveis de diferentes tipos de falhas. Naturalmente, estas devem ter em conta os custos das falhas. Você pode usar diferentes probabilidades para serviços de sistema diferentes. Você também pode gerar requisitos funcionais de confiabilidade. Novamente, isso pode ter de esperar até as decisões de projeto serem tomadas. No entanto, como discuto na Seção 12.3.2, em alguns casos é difícil criar especificações quantitativas. Você pode ser capaz de identificar apenas os requisitos funcionais de confiabilidade.

12.3.1 Métricas de confiabilidade

Em termos gerais, a confiabilidade pode ser especificada como a probabilidade de uma falha de sistema ocorrer quando um sistema estiver em uso dentro de um ambiente operacional especificado. Por exemplo, se você estiver disposto a aceitar que uma em cada mil operações possa falhar, então você pode especificar a probabilidade de falha como 0,001. Isso não significa, é claro, que a cada mil transações você terá uma falha. Significa que se você observar N mil transações, o número de falhas observadas deve ser em torno de N. Você pode refinar essa relação para diferentes tipos de falhas ou para diferentes partes do sistema. Você pode decidir que os componentes críticos devem ter uma probabilidade de falha menor do que os componentes não críticos.

Existem duas métricas importantes para especificar a confiabilidade, além de uma métrica adicional, usada para especificar os atributos de disponibilidade relacionados ao sistema. A escolha da métrica depende do tipo de sistema a ser especificado e os requisitos do domínio da aplicação. As métricas são:

1. *Probabilidade de falha sob demanda* (POFOD, do inglês *probability of failure on demand*). Se usar essa métrica, você definirá a probabilidade de uma demanda por serviços de um sistema resultar em uma falha de sistema. Assim, POFOD = 0,001 significa que existe 1/1.000 de chance de uma falha ocorrer quando surgir uma demanda.
2. *Taxa de ocorrência de falhas* (ROCOF, do inglês *rate of occurrence of failures*). Essa métrica define o provável número de falhas de sistema que podem ser observadas em relação a determinado período (por exemplo, uma hora), ou a um número de execuções de sistema. No exemplo anterior, a ROCOF é 1/1.000. A recíproca da ROCOF é o tempo médio para falha (MTTF, do inglês *mean time to failure*), que, por vezes, é usado como métrica de confiabilidade. O MTTF é o número médio de unidades de tempo entre falhas observadas no sistema. Portanto, uma ROCOF de duas falhas por hora implica tempo médio de 30 minutos entre cada falha.
3. *Disponibilidade* (AVAIL, do inglês *availability*). A disponibilidade de um sistema reflete sua capacidade de prestar serviços quando solicitado. AVAIL é a probabilidade de um sistema estar em operação quando surgir uma demanda por um serviço. Portanto, uma disponibilidade de 0,9999 significa que, em média, o sistema estará disponível em 99,99% do tempo em operação. A Tabela 12.3 mostra o que, na prática, diferentes níveis de disponibilidade significam.

A POFOD deve ser usada como uma métrica de confiabilidade em situações nas quais uma falha sob demanda pode levar a uma grave falha de sistema, o que se aplica independentemente da frequência das demandas. Por

Tabela 12.3 Especificação de disponibilidade

Disponibilidade	Explicação
0,9	O sistema está disponível 90% do tempo. Isso significa que, em um período de 24 horas (1.440 minutos), o sistema estará indisponível por 144 minutos.
0,99	Em um período de 24 horas, o sistema estará indisponível por 14,4 minutos.
0,999	O sistema estará indisponível por 84 segundos em um período de 24 horas.
0,9999	O sistema estará indisponível por 8,4 segundos em um período de 24 horas. Grosso modo, um minuto por semana.

exemplo, um sistema de proteção que monitora e desliga um reator químico caso a reação seja de superaquecimento deve ter sua confiabilidade especificada usando POFOD. Geralmente, as demandas de um sistema de proteção não são frequentes, pois o sistema é a última linha de defesa, após todas as outras estratégias de recuperação falharem. Portanto, uma POFOD de 0,001 (uma falha em mil demandas) pode parecer arriscada, mas, se houver apenas duas ou três demandas para o sistema em todo seu período de vida, então você provavelmente nunca verá uma falha do sistema.

A ROCOF é a métrica mais adequada para se usar em situações em que as demandas dos sistemas são feitas com regularidade e de forma não intermitente. Por exemplo, em um sistema que manipula um grande número de transações, você pode especificar uma ROCOF de dez falhas por dia. Isso significa que você está disposto a aceitar que, em média, não serão concluídas com êxito dez transações por dia, e estas terão de ser canceladas. Outra possibilidade é você especificar a ROCOF como o número de falhas a cada mil transações.

Se o tempo absoluto entre falhas é importante, você pode especificar a confiabilidade como o tempo médio entre falhas. Por exemplo, se você estiver especificando a confiabilidade requerida para um sistema com transações longas (como um sistema de projeto auxiliado por computador), você deve especificar a confiabilidade com um longo tempo médio para falha. O MTTF deve ser muito maior que o tempo médio que um usuário trabalha em seus modelos sem salvar os resultados. Isso significa que os usuários não seriam suscetíveis de perder o trabalho por meio de uma falha de sistema, em qualquer sessão.

Para avaliar a confiabilidade de um sistema, você precisa capturar dados sobre seu funcionamento. Os dados necessários podem incluir:

1. O número de falhas de sistema dado certo número de pedidos por serviços de sistema. Dessa forma, é possível medir a POFOD.
2. O tempo ou o número de transações entre as falhas de sistema, mais o tempo decorrido total ou o número total de transações. Dessa forma, é possível medir a ROCOF e o MTTF.
3. O tempo de reparação ou reinício após uma falha de sistema que ocasiona perda de serviço. É usado na medição de disponibilidade. A disponibilidade não depende apenas do tempo entre falhas, mas também do tempo necessário para o sistema voltar a funcionar.

As unidades de tempo que podem ser usadas são o tempo de calendário ou de processador, ou, ainda, uma unidade discreta, como o número de transações. Nos sistemas que gastam muito tempo aguardando para responder a uma solicitação por serviço, como os sistemas de comutação telefônica, a unidade de tempo que deve ser usada é o tempo de processador. Se você usar o de calendário, incluirá o período em que o sistema não estava fazendo nada.

Você deve usar o tempo de calendário para os sistemas que estão em operação contínua. Os sistemas de monitoramento, como sistemas de alarme e outros tipos de sistemas de controle de processos, enquadram-se nessa categoria. Sistemas que processam transações, como caixas eletrônicos de bancos ou sistemas de reservas aéreas, dependendo do horário do dia, têm cargas variáveis colocadas sobre eles. Nesses casos, a unidade de 'tempo' usada pode ser o número de transações (ou seja, a ROCOF seria o número de transações com falhas a cada N mil transações).

12.3.2 Requisitos não funcionais de confiabilidade

Os requisitos não funcionais de confiabilidade são especificações quantitativas de confiabilidade e disponibilidade requeridas de um sistema, calculadas por meio do uso de uma das métricas descritas na seção anterior. Especificações quantitativas de confiabilidade e disponibilidade têm sido usadas por muitos anos em sistemas de segurança críticos, mas raramente são usadas em sistemas de negócio críticos. No entanto, como cada vez mais empresas demandam serviços 24 horas por dia, sete dias por semana de seus sistemas, é provável que essas técnicas sejam cada vez mais usadas.

Existem várias vantagens em derivar especificações quantitativas de confiabilidade:

1. O processo de decidir o nível requerido de confiabilidade ajuda a esclarecer os *stakeholders* do que eles realmente precisam. Ajuda-os a compreender que existem diferentes tipos de falha de sistema e deixa claro que altos níveis de confiabilidade são muito caros para serem atingidos.
2. Fornece uma base para avaliar quando se deve parar de testar um sistema. Você para quando o sistema tiver atingido o nível requerido de confiabilidade.

3. É uma forma de avaliar diferentes estratégias de projeto destinadas a melhorar a confiabilidade de um sistema. Você pode fazer um julgamento sobre como cada estratégia pode conduzir aos níveis exigidos de confiabilidade.
4. Se o regulador tiver de aprovar um sistema antes de ele entrar em serviço (por exemplo, todos os sistemas que são críticos para a segurança de voo em uma aeronave são regulados), a evidência de que uma meta de confiabilidade requerida foi cumprida é importante para a certificação do sistema.

Para estabelecer o nível requerido de confiabilidade de sistema você precisa considerar as perdas associadas que poderiam resultar de uma falha de sistema. Essas perdas não são apenas financeiras, mas também a perda de reputação de uma empresa. A perda de reputação significa a perda de clientes. Embora no curto prazo as perdas resultantes de uma falha de sistema possam ser relativamente pequenas, a longo prazo elas podem ser muito mais significativas. Por exemplo, se você tentar acessar um site de comércio eletrônico e descobrir que ele não está disponível, você pode tentar encontrar o que deseja em outro lugar ao invés de esperar que o sistema se torne disponível. Se isso acontecer mais de uma vez, provavelmente você não vai comprar naquele site novamente.

O problema de especificar a confiabilidade usando métricas como POFOD, ROCOF e AVAIL é que é possível especificar em excesso a confiabilidade e, assim, incorrer em custos elevados de desenvolvimento e validação. A razão para isso é que os *stakeholders* acham difícil traduzir sua experiência prática em especificações quantitativas. Eles podem pensar que uma POFOD de 0,001 (1 falha em 1.000 demandas) representa um sistema relativamente confiável. No entanto, como já expliquei, se as demandas por um serviço são incomuns, na realidade, elas representam um nível muito elevado de confiabilidade.

Caso a confiabilidade seja especificada como uma métrica, obviamente será importante avaliar o nível requerido de confiabilidade alcançado. Faça com que essa avaliação seja parte dos testes de sistema. Para avaliar estatisticamente a confiabilidade de um sistema, você precisa observar uma série de falhas. Por exemplo, caso você tenha uma POFOD de 0,0001 (1 falha a cada 10 mil demandas), você pode ter de projetar testes que façam 50 mil ou 60 mil demandas em um sistema em que sejam observadas diversas falhas. Pode ser praticamente impossível projetar e implementar esse número de testes. Portanto, especificação excessiva de confiabilidade gera custos de teste muito altos.

Ao especificar a disponibilidade de um sistema, você pode ter problemas semelhantes. Apesar de um elevado nível de disponibilidade parecer desejável, a maioria dos sistemas tem padrões de demanda muito intermitentes (por exemplo, um sistema de negócios será usado, sobretudo, durante o horário comercial) e uma única figura de disponibilidade não reflete as necessidades do usuário. Quando o sistema está sendo usado, você precisa de alta disponibilidade, mas não em outros momentos. Naturalmente, dependendo do tipo de sistema, pode não haver diferença prática entre uma disponibilidade de 0,999 e uma disponibilidade de 0,9999.

Um problema fundamental da especificação excessiva é que pode ser praticamente impossível mostrar que um nível muito elevado de confiabilidade ou disponibilidade foi alcançado. Por exemplo, digamos que um sistema foi projetado para uso em uma aplicação crítica de segurança e, portanto, foi requerido que esta nunca falhe durante toda a vida útil do sistema. Suponha que mil cópias do sistema estão sendo instaladas e o sistema é executado mil vezes por segundo. A vida útil projetada do sistema é de dez anos. O número total de execuções do sistema é, portanto, cerca de 3×10^{14}. Não há sentido em especificar que a taxa de ocorrência de falhas deva ser de $1/10^{15}$ execuções (o que permite algum fator de segurança), pois você não pode testar o sistema por tempo suficiente para validar esse nível de confiabilidade.

Portanto, as organizações devem ser realistas sobre a necessidade de se especificar e validar um nível muito elevado de confiabilidade. Elevados níveis de confiabilidade são claramente justificados em sistemas nos quais operações confiáveis são críticas, como sistemas de comutação telefônica, ou em que falhas de sistema podem resultar em grandes perdas econômicas. Eles provavelmente não são justificados para muitos tipos de sistemas comerciais ou científicos. Estes sistemas têm modestos requisitos de confiabilidade, já que os custos de falha são apenas atrasos de processamento dos quais é relativamente simples e barato se recuperar.

Existe uma série de passos que você pode tomar para evitar a especificação excessiva de confiabilidade de sistema:

1. Especificar os requisitos de disponibilidade e confiabilidade para diferentes tipos de falhas. Deve haver uma menor probabilidade de ocorrência de falhas graves.
2. Especificar os requisitos de disponibilidade e confiabilidade de serviços separadamente. Falhas que afetam os serviços mais críticos devem ser especificadas como menos prováveis do que aquelas com efeitos locais, apenas. Você pode decidir limitar a especificação quantitativa de confiabilidade para os serviços de sistema mais críticos.

3. Decida se você realmente precisa de alta confiabilidade em um sistema de software ou se as metas de confiança gerais de sistema podem ser alcançadas de outras formas. Por exemplo, você pode usar mecanismos de detecção de erros para verificar as saídas de um sistema e dispor de processos para corrigir erros. Assim, pode não haver necessidade de um alto nível de confiabilidade no sistema que gera as saídas.

Para ilustrar esse último ponto, considere os requisitos de confiabilidade de um sistema de caixa eletrônico bancário (ATM) que dispensa dinheiro e oferece outros serviços aos clientes. Se houver problemas no hardware ou software do ATM, estes levarão a entradas incorretas no banco de dados da conta do cliente. Isso pode ser evitado, especificando-se um nível muito elevado de confiabilidade para o hardware e o software no ATM.

No entanto, os bancos têm muitos anos de experiência em identificar e corrigir as transações erradas de conta. Eles usam métodos de contabilidade para detectar quando as coisas dão errado. A maioria das transações que falham pode simplesmente ser cancelada, o que resulta em nenhuma perda para o banco e em pequena inconveniência para os clientes. Portanto, os bancos que controlam as redes de ATM aceitam que as falhas de ATM possam significar que um pequeno número de transações esteja incorreto, mas eles acreditam ser mais barato corrigir possíveis erros depois do que incorrer em custos muito elevados para evitar erros nas transações.

Para um banco (e para seus clientes), a disponibilidade da rede ATM é mais importante do que possíveis falhas em transações individuais ou não. A falta de disponibilidade significa maior demanda por serviços, insatisfação dos clientes, custos de engenharia para reparar a rede etc. Portanto, para sistemas baseados em transações, como sistemas bancários e de comércio eletrônico, geralmente o foco da especificação de confiabilidade está em especificar a disponibilidade do sistema.

Para especificar a disponibilidade de uma rede ATM, você deve identificar os serviços de sistema e especificar a disponibilidade necessária para cada um deles. São eles:

- o serviço de banco de dados da conta do cliente;
- os serviços individuais fornecidos por uma rede ATM, como 'retirada de dinheiro', 'fornecer informações sobre a conta' etc.

Nesse caso, o serviço de banco de dados é mais crítico, pois as falhas desse serviço significam que todas as ATMs na rede estão fora de ação. Portanto, para ter um alto nível de disponibilidade, você precisa dessa especificação. Nesse caso, um valor aceitável para a disponibilidade do banco de dados (ignorando questões como manutenção programada e atualizações) provavelmente seria de cerca de 0,9999, entre sete horas da manhã e 23 horas, o que significa períodos de inatividade de menos de um minuto por semana. Na prática, significa que poucos clientes seriam afetados e seriam causadas inconveniências leves para o consumidor.

Para uma ATM individual, a disponibilidade global depende da confiabilidade mecânica e do fato de ela poder funcionar mesmo sem dinheiro. Problemas de software tendem a ter menos efeito do que fatores como esses. Assim, é aceitável um menor nível de disponibilidade para o software de ATM. A disponibilidade global do software de ATM pode ser especificada como 0,999, o que significa que uma máquina pode ficar indisponível por um ou dois minutos a cada dia.

Para ilustrar a especificação de confiabilidade baseada em falhas, considere os requisitos de confiabilidade para o software de controle da bomba de insulina. Esse sistema libera insulina certo número de vezes por dia e monitora, várias vezes por hora, a glicose no sangue do usuário, durante horas. Como o uso do sistema é intermitente e as consequências de falha são graves, a métrica mais adequada de confiabilidade é a POFOD (probabilidade de falha sob demanda).

Existem dois tipos de falha possíveis na bomba de insulina:

1. Falhas transitórias de software, que podem ser reparadas por ações do usuário, como reiniciar ou recalibrar a máquina. Para esses tipos de falhas, um valor relativamente baixo de POFOD (digamos 0,002) pode ser aceitável. Isso significa que, a cada 500 demandas colocadas na máquina, pode ocorrer uma falha, ou seja, aproximadamente uma vez a cada 3,5 dias, pois o açúcar no sangue é avaliado cerca de cinco vezes por hora.
2. Falhas permanentes de software que exigem que ele seja reinstalado pelo fabricante. A probabilidade desse tipo de falha deve ser muito menor. Grosso modo, uma vez por ano é o valor mínimo, de modo que a POFOD não deve ser mais do que 0,00002.

No entanto, falhas na liberação de insulina não têm implicações imediatas para a segurança, então, mais do que os fatores de segurança, os fatores comerciais regem o nível de confiabilidade requerido. Os custos de serviços são elevados, pois os usuários necessitam de reparação e substituição rápidas. É do interesse do fabricante limitar o número de falhas permanentes que requerem reparação.

12.3.3 Especificação funcional de confiabilidade

Especificações funcionais de confiabilidade envolvem a identificação dos requisitos que definem restrições e características que contribuem para a confiabilidade de sistema. Para sistemas nos quais a confiabilidade tenha sido especificada quantitativamente, esses requisitos funcionais podem ser necessários para assegurar que um nível requerido de confiabilidade seja alcançado.

Existem três tipos de requisitos funcionais de confiabilidade para um sistema:

1. *Requisitos de verificação*. Identificam verificações de entradas do sistema para garantir que entradas incorretas ou fora dos limites sejam detectadas antes de serem processadas pelo sistema.
2. *Requisitos de recuperação*. São orientados para ajudar o sistema a se recuperar de falhas. Normalmente, esses requisitos estão relacionados com a manutenção de cópias do sistema e de seus dados e com a especificação de como restaurar os serviços de sistema após uma falha.
3. *Requisitos de redundância*. Especificam as características redundantes do sistema que garantem que a falha de um único componente não causa a perda do serviço completo. Esse tema é discutido mais detalhadamente no Capítulo 13.

Além disso, os requisitos de confiabilidade podem incluir requisitos de processo para a confiabilidade. Esses são os requisitos para garantir que boas práticas, conhecidas por reduzir o número de defeitos em um sistema, sejam usadas no processo de desenvolvimento. Alguns exemplos de requisitos funcionais de confiabilidade e de processo são mostrados no Quadro 12.2.

Não há regras simples para derivação de requisitos funcionais de confiabilidade. Nas organizações que desenvolvem sistemas críticos, geralmente existe conhecimento organizacional sobre os possíveis requisitos de confiabilidade e sobre como eles afetam a confiabilidade de um sistema real. Essas organizações podem especializar-se em tipos específicos de sistema, como sistemas de controle de ferrovia, nos quais os requisitos de confiabilidade podem ser reusados em uma série de sistemas.

12.4 Especificação de proteção

A especificação de requisitos de proteção para sistemas tem algo em comum com os requisitos de segurança. É impraticável especificá-los quantitativamente, e, muitas vezes, os requisitos de proteção são requisitos do tipo 'não deve' que definem os comportamentos inaceitáveis do sistema, em vez de definir a funcionalidade requerida. No entanto, por uma série de razões, a proteção é um problema mais desafiador do que a segurança:

1. Ao considerar a segurança, você pode supor que o ambiente no qual o sistema está instalado não é hostil. Ninguém está tentando provocar um incidente relacionado à segurança. Ao considerar a proteção, você precisa assumir que os ataques ao sistema são deliberados e que o invasor talvez tenha conhecimento de pontos fracos do sistema.
2. Quando ocorrem falhas de sistema que representam um risco para a segurança, você procura os erros ou omissões que causaram a falha. Quando os ataques deliberados causam falhas de sistema, encontrar a causa-raiz pode ser mais difícil, pois o invasor pode tentar esconder a causa da falha.
3. É geralmente aceitável desligar um sistema ou degradar os serviços de sistema para evitar uma falha de segurança. No entanto, os ataques em um sistema podem ser chamados ataques de negação de serviços, os quais se destinam a desligar o sistema. Desligar o sistema significa que o ataque foi bem-sucedido.

Quadro 12.2 Exemplos de requisitos de confiabilidade funcionais (RC, do inglês *reliability requirements*)

RC1: Um intervalo predefinido deve ser estabelecido para todas as entradas do operador e o sistema verificará se todas as entradas do operador ficam dentro desse intervalo predefinido. (Verificação)
RC2: Cópias da base de dados de pacientes devem ser mantidas em dois servidores separados, não alojados no mesmo edifício. (Recuperação, redundância)
RC3: Programação N-*version* deve ser usada para implementar o sistema de controle de frenagem. (Redundância)
RC4: O sistema deve ser implementado em um subconjunto seguro de Ada e controlado por meio de análise estática. (Processo)

4. Eventos de segurança não são gerados por um adversário inteligente. Um invasor pode sondar as defesas de um sistema em uma série de ataques, modificando os ataques conforme aprenda mais sobre o sistema e suas respostas.

Essas distinções geralmente significam que os requisitos de proteção precisam ser mais extensos do que os requisitos de segurança. Estes, por sua vez, geram requisitos funcionais do sistema que fornecem proteção contra eventos e defeitos, que podem causar falhas de segurança. Esses requisitos estão mais preocupados em verificar os problemas e em tomar medidas no caso de problemas. Em contrapartida, existem muitos tipos de requisitos de proteção que cobrem as diferentes ameaças enfrentadas pelo sistema. Firesmith (2003) identificou dez tipos de requisitos de proteção que podem ser incluídos em uma especificação do sistema:

1. Os requisitos de identificação anunciam se um sistema deve identificar seus usuários antes de interagir com eles.
2. Os requisitos de autenticação especificam como os usuários são identificados.
3. Os requisitos de autorização especificam os privilégios e as permissões de acesso dos usuários identificados.
4. Os requisitos de imunidade especificam como um sistema deve se proteger contra vírus, *worms* e outras ameaças.
5. Os requisitos de integridade especificam como evitar a corrupção de dados.
6. Os requisitos de detecção de intrusão especificam quais mecanismos devem ser usados na detecção de ataques ao sistema.
7. Os requisitos de não repúdio especificam que, quando em uma transação, uma parte não pode negar sua participação nessa transação.
8. Os requisitos de privacidade especificam como a privacidade de dados deve ser mantida.
9. Os requisitos de auditoria de proteção especificam como o uso do sistema pode ser auditado e verificado.
10. Os requisitos de proteção de manutenção de sistema especificam como uma aplicação pode impedir que as mudanças autorizadas comprometam, acidentalmente, seus mecanismos de proteção.

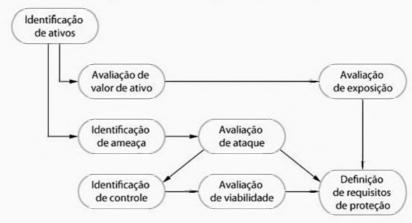
Claro que você não vai ver todos esses tipos de requisitos de proteção em todos os sistemas. Os requisitos particulares dependem do tipo de sistema, da situação de uso e dos usuários esperados.

O processo de análise e avaliação de riscos discutido na Seção 12.1 pode ser usado para identificar os requisitos de proteção de sistema. Como já discutimos, existem três estágios para esse processo:

1. *Análise preliminar de riscos*. Nesse estágio, não foram tomadas as decisões sobre os requisitos detalhados de sistema, o projeto de sistema ou a tecnologia de implementação. O objetivo desse processo de avaliação é derivar os requisitos de proteção para o sistema como um todo.
2. *Análise de riscos de ciclo de vida*. Essa avaliação ocorre durante o ciclo de vida de desenvolvimento de sistema após serem feitas as escolhas de projeto. Os requisitos de proteção adicionais levam em conta as tecnologias usadas na construção do sistema e nas decisões de projeto e implementação de sistema.
3. *Análise de riscos operacionais*. Essa avaliação considera os riscos decorrentes de ataques maliciosos ao sistema operacional, feitos pelos usuários, com ou sem conhecimento interno do sistema.

Os processos de avaliação e análise de riscos usados na especificação de requisitos de proteção são variantes do processo genérico de especificação dirigido a riscos discutido na Seção 12.1. Um processo de requisitos de proteção dirigido a riscos é mostrado na Figura 12.4. Esse pode parecer diferente do processo dirigido a riscos da Figura 12.1, mas, colocando a atividade do processo genérico entre parênteses, eu indico como cada estágio corresponde a diferentes estágios do processo genérico. Os estágios do processo são:

1. Identificação de ativos, em que são identificados os ativos de sistema que podem exigir proteção. O próprio sistema ou as funções de sistema em particular podem ser identificados como ativos, bem como os dados associados ao sistema (identificação de riscos).
2. Avaliação de valor de ativo, em que você estima o valor dos ativos identificados (análise de riscos).
3. Avaliação de exposição, em que você avalia as potenciais perdas associadas a cada ativo. Isso deve levar em conta as perdas diretas, como roubo de informações, custos de cobrança e possível perda de reputação (análise de riscos).
4. Identificação de ameaça, em que você identifica as ameaças aos ativos do sistema (análise de riscos).
5. Avaliação de ataque, em que você decompõe cada ameaça em ataques que poderiam ser feitos ao sistema

Figura 12.4 O processo de avaliação preliminar de riscos para requisitos de proteção

e as possíveis maneiras de esses ataques ocorrerem. Você pode usar as árvores de ataques (SCHNEIER, 1999) para analisar os possíveis ataques. Estas são semelhantes às árvores de defeitos; você começa com uma ameaça na raiz da árvore e identifica os possíveis ataques causais e como eles poderiam acontecer (decomposição de riscos).

6. Identificação de controle, em que você propõe os controles que podem ser colocados em prática para proteger um ativo. Os controles são mecanismos técnicos, como a criptografia, que podem ser usados para proteger os ativos (redução de riscos).
7. Avaliação de viabilidade, em que você avalia a viabilidade técnica e os custos dos controles propostos. Não vale a pena ter controles caros para proteger ativos que não têm valores altos (redução de riscos).
8. Definição de requisitos de proteção, em que o conhecimento da exposição, ameaças e avaliações de controle são usados para derivar os requisitos de proteção do sistema. Estes podem ser requisitos para a infraestrutura de sistema ou para o sistema de aplicação (redução de riscos).

Uma informação importante para os processos de avaliação e gerenciamento de riscos diz respeito à política de proteção organizacional. Uma política de proteção organizacional aplica-se a todos os sistemas e define o que deve e o que não deve ser permitido. Por exemplo, um aspecto de uma política de proteção militar pode indicar que 'Os leitores só podem examinar documentos cuja classificação seja a mesma ou abaixo do nível de habilitação do leitor'. Isso significa que se o leitor estiver no nível 'secreto', ele poderá ter acesso a documentos classificados como 'secretos', 'confidenciais' ou 'abertos', mas não a documentos classificados como 'ultrassecretos'.

A política de proteção estabelece as condições que sempre devem ser mantidas por um sistema de proteção e ajuda a identificar ameaças que possam surgir. As ameaças são quaisquer coisas que possam ameaçar a proteção do negócio. Na prática, as políticas de proteção geralmente são documentos informais que definem o que é permitido e o que não é. No entanto, Bishop (2005) discute a possibilidade de expressar as políticas de proteção em uma linguagem formal e a criação de verificações automatizadas para assegurar que a política esteja sendo seguida.

Para ilustrar esse processo da análise de riscos de segurança, considere o sistema de informação hospitalar para cuidados de saúde mental, MHC-PMS. Neste livro, não tenho espaço para discutir uma avaliação de riscos completa, mas posso recorrer a esse sistema como uma fonte de exemplos. Tenho apresentado esses exemplos como fragmentos de um relatório (tabelas 12.4 e 12.5), o qual pode ser gerado a partir do processo preliminar de avaliação de riscos. Esse relatório de análise preliminar de riscos é usado para definir os requisitos de proteção.

A partir da análise de riscos para o sistema de informação hospitalar você pode derivar os requisitos de proteção. Alguns exemplos desses requisitos são:

1. No início de uma consulta médica, as informações sobre o paciente devem ser carregadas do banco de dados para uma área segura de cliente de sistema.
2. Todas as informações sobre o paciente que estiverem no sistema do cliente devem ser criptografadas.
3. As informações sobre o paciente devem ser transferidas para o banco de dados quando uma consulta médica termina e é excluída do computador do cliente.

Tabela 12.4 Análise de ativos em um relatório preliminar de avaliação de riscos para o MHC-PMS

Ativo	Valor	Exposição
O sistema de informação	Alto. Necessário para suportar todas as consultas clínicas. Potencialmente crítico de segurança.	Alta. Perdas financeiras à medida que consultas podem ter de ser canceladas. Custos de restauração de sistema. Possível dano ao paciente caso o tratamento não possa ser prescrito.
O banco de dados de pacientes	Alto. Necessário para suportar todas as consultas clínicas. Potencialmente crítico de segurança.	Alta. Perdas financeiras à medida que consultas podem ter de ser canceladas. Custos de restauração de sistema. Possível dano ao paciente caso o tratamento não possa ser prescrito.
Um registro individual de paciente	Normalmente baixo, embora possa ser elevado para determinados pacientes, dependendo do perfil.	Perdas diretas baixas, mas uma possível perda de reputação.

4. Um registro de todas as alterações feitas no banco de dados de sistema e o iniciador dessas mudanças devem ser mantidos em um computador separado do servidor de banco de dados.

Os dois primeiros requisitos estão relacionados — as informações sobre o paciente são transferidas para uma máquina local para que as consultas possam continuar caso o servidor do banco de dados do paciente seja atacado ou fique indisponível. No entanto, essa informação deve ser excluída, assim outros usuários do computador cliente não terão acesso a elas. O quarto requisito é um requisito de recuperação e auditoria, o que significa que as mudanças podem ser recuperadas ao passarem pelas alterações no registro e que é possível descobrir quem fez as alterações. Essa responsabilização desencoraja o uso indevido do sistema por pessoal autorizado.

12.5 Especificação formal

Por mais de 30 anos, muitos pesquisadores têm defendido o uso de métodos formais de desenvolvimento de software. Os métodos formais são abordagens baseadas em matemática para o desenvolvimento de software, em que é definido um modelo formal do software. É possível analisar formalmente esse modelo e usá-lo como base para uma especificação formal de sistema. Em princípio, é possível começar com um modelo formal de software e provar que o programa desenvolvido é consistente com o modelo, eliminando-se, assim, falhas de software resultantes de erros de programação.

O ponto de partida para todos os processos formais de desenvolvimento é um modelo formal de sistema, que serve como uma especificação de sistema. Para criar esse modelo, você traduz os requisitos de usuário do sistema,

Tabela 12.5 Análise de ameaça e controle em um relatório preliminar de avaliação de riscos

Ameaça	Probabilidade	Controle	Viabilidade
Usuário não autorizado ganha acesso como administrador de sistema e torna o sistema indisponível	**Baixa**	Somente permitir a administração de sistema a partir de locais específicos, fisicamente protegidos.	Baixo custo de implementação, mas é preciso ter cuidado com a distribuição de chaves, para garantir que elas estejam disponíveis em caso de emergência.
Usuário não autorizado ganha acesso como usuário de sistema e acessa informações confidenciais	**Alta**	Exigir autentificação de todos os usuários, por meio de um mecanismo biométrico. Registrar todas as alterações nas informações do paciente para acompanhar o uso do sistema.	Tecnicamente possível, mas o custo seria muito alto. Possível resistência de usuários. Implementação simples e transparente, também suporta a recuperação.

expressos em linguagem natural, diagramas e tabelas, em uma linguagem matemática que define formalmente a semântica. A especificação formal é uma descrição inequívoca do que o sistema deve fazer. Usando métodos manuais ou apoiados por ferramentas, é possível verificar se o comportamento de um programa é consistente com a especificação.

As especificações formais não são apenas essenciais para uma verificação do projeto e implementação do software. Elas são a maneira mais precisa de especificação dos sistemas e, assim, de redução da possibilidade de mal-entendidos. Além disso, a construção de uma especificação formal força uma análise detalhada dos requisitos, e essa é uma maneira eficaz de descobrir problemas de requisitos. Em uma especificação em linguagem natural, os erros podem ser ocultados pela imprecisão da linguagem. Esse não é o caso quando o sistema é especificado formalmente.

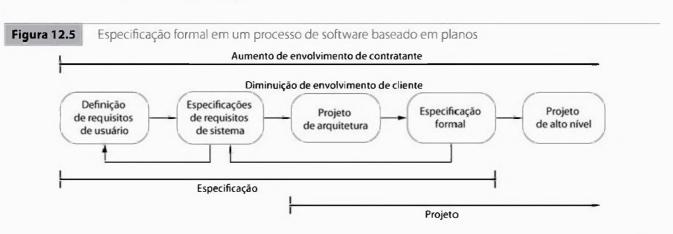
Geralmente, as especificações formais são desenvolvidas como parte de um processo de software baseado em planos, no qual o sistema é completamente especificado antes do desenvolvimento. Os requisitos e o projeto de sistema são definidos em detalhes e são cuidadosamente analisados e verificados antes do início da implementação. Se a especificação formal do software é desenvolvida, isso geralmente acontece depois de os requisitos de sistema serem especificados, mas antes do projeto detalhado de sistema. Existe um ciclo curto de realimentação entre a especificação detalhada e a especificação formal de requisitos.

A Figura 12.5 mostra os estágios de especificação de software e sua interface com o projeto de software em um processo baseado em planos. Como é caro desenvolver especificações formais, você pode decidir limitar o uso dessa abordagem para os componentes críticos para o funcionamento do sistema. Você identifica esses componentes no projeto de arquitetura do sistema.

Ao longo dos últimos anos, foi desenvolvido o suporte automatizado para análise de uma especificação formal. Os verificadores de modelos (CLARKE et al., 2000) são uma ferramenta de software com uma especificação formal baseada em estados (um modelo de sistema) como uma entrada, junto com a especificação de algumas propriedades desejáveis formalmente expressas, como 'não existem estados inalcançáveis'. O programa de verificação de modelos analisa exaustivamente a especificação e relata que a propriedade do sistema é satisfeita pelo modelo ou apresenta um exemplo que mostra que esta não é satisfeita. A verificação de modelos está intimamente relacionada com a noção de análise estática. No Capítulo 15, discuto essas abordagens gerais para verificação de sistema.

As vantagens de se desenvolver uma especificação formal e usar essa especificação em um processo formal de desenvolvimento são:

1. Ao desenvolver uma especificação formal em detalhes, você desenvolve uma compreensão profunda e detalhada dos requisitos de sistema. Mesmo se você não usar a especificação em um processo formal de desenvolvimento, a detecção de erros de requisitos é um argumento poderoso para o desenvolvimento de uma especificação formal (HALL, 1990). Geralmente, os problemas de requisitos descobertos no início são muito mais baratos de serem corrigidos do que quando se encontram em estágios posteriores do processo de desenvolvimento.
2. Como a especificação é expressa em uma linguagem com semântica formalmente definida, você pode analisá-la automaticamente para descobrir incoerências e incompletude.
3. Se você usar o método B, por exemplo, poderá transformar a especificação formal em um programa por meio de uma sequência de transformações de correção/preservação. Portanto, é garantido que o programa resultante atenderá a sua especificação.

Figura 12.5 Especificação formal em um processo de software baseado em planos

4. Os custos de testes de programa podem ser reduzidos porque você verificou o programa e sua especificação.

Apesar dessas vantagens, os métodos formais tiveram um impacto limitado no desenvolvimento prático de software, mesmo para sistemas críticos. Consequentemente, existe muito pouca experiência na comunidade sobre desenvolvimento e uso de especificação formal de sistema. Os argumentos apresentados contra o desenvolvimento dessa especificação formal são:

1. Os donos do problema e especialistas em domínio podem não entender uma especificação formal, então não podem verificar se ela representa precisamente seus requisitos. Os engenheiros de software, que compreendem a especificação formal podem não entender o domínio de aplicação, de modo que eles também não podem ter certeza de que a especificação formal é um reflexo exato dos requisitos de sistema.
2. É bastante fácil quantificar os custos da criação de uma especificação formal, mas mais difícil estimar a possível economia resultante de seu uso. Como resultado, os gerentes não estão dispostos a assumir o risco de adotar essa abordagem.
3. A maioria dos engenheiros de software não foi treinada para usar linguagens formais de especificação. Assim, eles relutam em propor seu uso em processos de desenvolvimento.
4. É difícil escalar abordagens atuais para especificações formais até os sistemas muito grandes. Sobretudo quando a especificação formal é usada para a especificação de software crítico de Kernel em vez de sistemas completos.
5. Especificação formal não é compatível com os métodos ágeis de desenvolvimento.

No entanto, quando este livro foi escrito, os métodos formais foram usados no desenvolvimento de uma série de aplicações críticas de segurança e de proteção. Eles também podem ser usados de maneira efetiva no desenvolvimento e validação de partes críticas de sistemas de software grandes e mais complexos (BADEAU e AMELOT, 2005; HALL, 1996; HALL e CHAPMAN, 2002; MILLER et al., 2005; WORDWORTH, 1996). Eles são a base de ferramentas usadas na verificação estática, como o sistema de verificação de *driver* usado pela Microsoft (BALL et al., 2004; BALL et al., 2006) e a linguagem SPARK/Ada (BARNES, 2003) para a engenharia de sistemas críticos.

PONTOS IMPORTANTES

- A análise de risco é uma atividade importante na especificação de requisitos de proteção e confiança. Envolve identificar os riscos que podem resultar em acidentes ou incidentes. Requisitos de sistema são, então, gerados para garantir que esses riscos não ocorram e, caso aconteçam, que eles não ocasionem um incidente ou acidente.
- Uma abordagem orientada a perigos pode ser usada para entender os requisitos de segurança para um sistema. Você identifica os perigos potenciais e os decompõe (com métodos como a análise da árvore de defeitos) para descobrir suas causas. Então, você especifica os requisitos para evitar ou se recuperar desses problemas.
- Na especificação de requisitos de sistema, os requisitos de confiabilidade podem ser definidos quantitativamente. As métricas de confiabilidade incluem a probabilidade de falha sob demanda (POFOD), a taxa de ocorrência de falhas (ROCOF) e a disponibilidade (AVAIL).
- É importante não superespecificar a confiabilidade requerida do sistema, pois isso pode gerar custos adicionais desnecessários nos processos de desenvolvimento e validação.
- Requisitos de proteção são mais difíceis de identificar do que os de segurança, porque um invasor pode usar o conhecimento sobre as vulnerabilidades do sistema para planejar um ataque a ele, e pode aprender sobre as vulnerabilidades de ataques malsucedidos.
- Para especificar os requisitos de proteção, você deve identificar os ativos que devem ser protegidos e definir como as técnicas de proteção e tecnologia devem ser usadas para protegê-los.
- Métodos formais de desenvolvimento de software dependem de uma especificação de sistema, expressa como um modelo matemático. Desenvolver uma especificação formal tem o benefício fundamental de estimular um exame e análise detalhados dos requisitos de sistema.

Safeware: System Safety and Computers. Essa é uma discussão aprofundada de todos os aspectos de sistemas de segurança críticos. É particularmente forte em sua descrição da análise de perigos e da derivação de requisitos a partir deles. (LEVESON, N. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.)

'Security Use Cases.' Um bom artigo, disponível na Internet, que se concentra em como os casos de uso podem ser usados na especificação de proteção. O autor também tem uma série de bons artigos sobre as especificações de proteção citadas nesse artigo. (FIRESMITH, D. G. *Journal of Object Technology*, v. 2, n. 3, mai.-jun. 2003.) Disponível em: <http://www.jot.fm/issues/issue_2003_05/column6/>.

'Ten Commandments of Formal Methods... Ten Years Later.' Esse é um conjunto de diretrizes para o uso de métodos formais, proposto pela primeira vez em 1996 e que é revisitado no presente artigo. É um bom resumo das questões práticas em torno do uso de métodos formais. (BOWEN, J. P.; HINCHEY, M. G. *IEEE Computer*, v. 39, n. 1, jan. 2006.) Disponível em: <http://dx.doi.org/10.1109/MC.2006.35>.

'Security Requirements for the Rest of Us: A Survey.' Um bom ponto de partida para a leitura sobre a especificação de requisitos de proteção. Os autores focam abordagens leves em vez de abordagens formais. (TØNDEL, I. A.; JAATUN, M. G.; MELAND, P. H. *IEEE Software*, v. 25, n. 1, jan./fev. 2008.) Disponível em: <http://dx.doi.org/10.1109/MS.2008.19>.

12.1 Explique por que os limites do triângulo de risco mostrados na Figura 12.5 são suscetíveis a sofrer alteração com o tempo e mudanças de atitudes sociais.

12.2 Explique por que, ao especificar a segurança e a proteção, a abordagem baseada em riscos é interpretada de diferentes maneiras.

12.3 No sistema de bomba de insulina, o usuário deve trocar a agulha e o fornecimento de insulina em intervalos regulares, e também pode alterar a dose única máxima e a dose máxima diária que podem ser administradas. Sugira três erros de usuário que podem ocorrer e proponha requisitos de segurança capazes de evitar que esses erros resultem em um acidente.

12.4 Um sistema de software crítico de segurança para o tratamento de pacientes com câncer tem dois componentes principais:

- Uma máquina de radioterapia que provê doses controladas de radiação para as regiões de tumor. Essa máquina é controlada por um sistema de software embutido.
- Um banco de dados de tratamento que inclui detalhes sobre o tratamento de cada paciente. Os requisitos de tratamento são inseridos nesse banco de dados e automaticamente transferidos para a máquina de radioterapia.

Identifique três perigos que podem surgir nesse sistema. Para cada perigo, sugira um requisito de defesa que reduzirá a probabilidade de esses perigos resultarem em um acidente. Explique por que sua defesa sugerida poderá reduzir o risco associado ao perigo.

12.5 Sugira métricas de confiabilidade adequadas para as classes de sistemas de software a seguir. Justifique sua escolha de métricas. Preveja o uso desses sistemas e sugira os valores apropriados para as métricas de confiabilidade.

- Um sistema que monitora pacientes em uma unidade de terapia intensiva de um hospital.
- Um editor de texto.
- Um sistema de controle automatizado de *vending machine*.
- Um sistema de controle de frenagem em um carro.
- Um sistema para controlar uma unidade de refrigeração.
- Um gerador de relatório de gerenciamento.

12.6 Um sistema de proteção de trens aciona os freios de um trem automaticamente caso o limite de velocidade para um segmento de via seja ultrapassado ou caso o trem entre em um segmento de via que esteja

sinalizado com uma luz vermelha (ou seja, o caminho não deve ser tomado). Fundamentando sua resposta, escolha uma métrica de confiabilidade que possa ser usada para especificar a confiabilidade requerida para tal sistema.

12.7 Existem dois requisitos de segurança essenciais para o sistema de proteção de trem:

- O trem não deve entrar em um segmento de via sinalizado com uma luz vermelha.
- O trem não pode exceder o limite de velocidade estabelecido para um segmento de via.

Supondo que o *status* de sinal e o limite de velocidade para o segmento de via sejam transmitidos para o software a bordo no trem antes de ele entrar no segmento de via, proponha cinco possíveis requisitos funcionais de sistema para o software a bordo que possam ser gerados a partir dos requisitos de segurança de sistema.

12.8 Explique por que, durante o desenvolvimento de um sistema, existe a necessidade da avaliação preliminar de riscos de proteção e da avaliação de riscos de proteção de ciclo de vida.

12.9 A partir da Tabela 12.5, identifique duas novas ameaças ao MHC-PMS, junto com seus controles associados. Use-as como base para gerar requisitos de proteção de software adicionais para implementar os controles propostos.

12.10 Os engenheiros de software que trabalham na especificação e desenvolvimento de sistemas de segurança devem ser profissionais certificados? Explique seu raciocínio.

REFERÊNCIAS

BADEAU, F.; AMELOT, A. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. *Proc. ZB 2005*: Formal Specification and Development in Z and B. Guildford, Reino Unido: Springer, 2005.

BALL, T.; BOUNIMOVA, E.; COOK, B.; LEVIN, V.; LICHTENBERG, J.; McGARVEY, C. et al. Thorough Static Analysis of Device Drivers. *Proc. EuroSys 2006*. Leuven, Bélgica, 2006.

BALL, T.; COOK, B.; LEVIN, V.; RAJAMANI, S. K. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods Inside Microsoft. *Proc. Integrated Formal Methods 2004*. Canterbury, Reino Unido: Springer, 2004.

BARNES, J. P. *High-integrity Software: The SPARK Approach to Safety and Security*. Harlow, Reino Unido: Addison-Wesley, 2003.

BISHOP, M. *Introduction to Computer Security*. Boston: Addison-Wesley, 2005.

BRAZENDALE, J.; BELL, R. Safety-related control and protection systems: standards update. *IEE Computing and Control Engineering J.*, v. 5, n. 1, 1994, p. 6-12.

CLARKE, E. M.; GRUMBERG, O.; PELED, D. A. *Model Checking*. Cambridge, Massassuchetts.: MIT Press, 2000.

FIRESMITH, D. G. Engineering Security Requirements. *Journal of Object Technology*, v. 2, n. 1, 2003, p. 53-68.

HALL, A. Seven Myths of Formal Methods. *IEEE Software*, v, 7, n. 5, 1990, p. 11-20.

______. Using Formal methods to Develop an ATC Information System. *IEEE Software*, v. 13, n. 2, 1996, p. 66-76.

HALL, A.; CHAPMAN, R. Correctness by Construction: Developing a Commercially Secure System. *IEEE Software*, v. 19, n. 1, 2002, p. 18-25.

JAHANIAN, F.; MOK, A. K. Safety analysis of timing properties in real-time systems. *IEEE Trans.on Software Engineering*, v. SE-12, n. 9, 1986, p. 890-904.

LEVESON, N.; STOLZY, J. Safety analysis using Petri nets. *IEEE Transactions on Software Engineering*, v. 13, n. 3, 1987, p. 386-397.

LEVESON, N. G. *Safeware: System Safety and Computers*. Reading, Mass.: Addison-Wesley, 1995.

MILLER, S. P.; ANDERSON, E. A.; WAGNER, L. G.; WHALEN, M. W.; HEIMDAHL, M. P. E. Formal Verification of Flight Control Software. *Proc. AIAA Guidance*, Navigation and Control Conference, San Francisco, 2005.

PETERSON, J. L. *Petri Net Theory and the Modeling of Systems*. Nova York: McGraw-Hill, 1981.

SCHNEIER, B. Attack Trees. *Dr Dobbs Journal*, v. 24, n. 12, 1999, p. 1-9.

STOREY, N. *Safety-Critical Computer Systems*. Harlow, Reino Unido: Addison-Wesley, 1996.

WORDSWORTH, J. *Software Engineering with B*. Wokingham: Addison-Wesley, 1996.

CAPÍTULO

13

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Engenharia de confiança

Objetivos

O objetivo deste capítulo é discutir os processos e técnicas de desenvolvimento de sistemas altamente confiáveis. Com a leitura deste capítulo, você:

- entenderá como a confiança de sistema pode ser obtida por meio de componentes redundantes e diversificados;
- saberá como os processos confiáveis de software contribuem para o desenvolvimento de softwares confiáveis;
- entenderá como os diferentes estilos de arquitetura podem ser usados para implementar redundância e diversidade de software;
- estará ciente de boas práticas de programação que devem ser usadas em engenharia de sistemas confiáveis.

Conteúdo

O uso de técnicas de engenharia de software, as melhores linguagens de programação e o melhor gerenciamento de qualidade têm levado a melhorias significativas na confiança da maioria dos softwares. No entanto, ainda podem ocorrer falhas de sistema que afetam sua disponibilidade ou conduzem à produção de resultados incorretos. Em alguns casos, essas falhas causam pequenos incômodos. Os vendedores de sistemas podem simplesmente decidir conviver com essas falhas, sem corrigir os erros em seus sistemas. No entanto, em alguns sistemas, uma falha pode causar a perda de vidas, ou perdas econômicas significativas, bem como de reputação. Esses são conhecidos como 'sistemas críticos', para os quais o alto nível de confiança é essencial.

Exemplos de sistemas críticos incluem sistemas de controle de processos, sistemas de proteção que desligam outros sistemas em caso de falha, sistemas médicos, comutadores de telecomunicações e sistemas de controle de voo. As ferramentas e técnicas especiais de desenvolvimento podem melhorar a confiança do software em um sistema crítico. Normalmente, essas ferramentas e técnicas aumentam os custos de desenvolvimento de sistema, mas reduzem o risco de falhas e as perdas que podem resultar dessas falhas.

A engenharia de confiança está preocupada com as técnicas para aumentar a confiança de ambos os sistemas, críticos e não críticos. Essas técnicas suportam três abordagens complementares que são usadas no desenvolvimento de softwares confiáveis.

1. *Prevenção de defeitos*. O processo de projeto e implementação de software deve usar abordagens de desenvolvimento de software que ajudem a evitar erros de projeto e programação e, assim, minimizar o número de defeitos que possam surgir quando o sistema estiver em execução. Poucos defeitos significam menores chances de falhas durante a execução.
2. *Detecção e correção de defeitos*. Os processos de verificação e validação são projetados para descobrir e remover defeitos em um programa, antes que este seja implantado para uso operacional. Sistemas críticos exigem extensa

verificação e validação para se descobrir o maior número possível de defeitos antes da implantação e para convencer os *stakeholders* de que o sistema é confiável. No Capítulo 15, eu abordo esse tema.

3. *Tolerância a defeitos*. O sistema é projetado de modo que os defeitos ou o comportamento inesperado sejam detectados durante a execução e é gerenciado de forma que não ocorra a falha de sistema. As abordagens simples de tolerância a defeitos, com base em verificação interna durante a execução, podem ser incluídas em todos os sistemas. Entretanto, as técnicas mais especializadas de tolerância a defeitos (como o uso de arquiteturas de sistema tolerantes a defeitos) são geralmente usadas quando é necessário alto nível de disponibilidade e confiabilidade de sistema.

Infelizmente, aplicar técnicas de prevenção, de detecção e de tolerância a defeitos ocasiona retornos decrescentes. Os custos de se encontrar e remover os defeitos remanescentes em um sistema de software aumentam exponencialmente à medida que os defeitos de programa são descobertos e removidos (Figura 13.1). Como o software se torna mais confiável, você precisa gastar mais tempo e mais esforço para encontrar cada vez menos defeitos. Em algum momento, mesmo para sistemas críticos, os custos desse esforço adicional tornam-se injustificáveis.

Como resultado, as empresas de desenvolvimento de software aceitam que os softwares sempre conterão alguns defeitos residuais. O nível de defeitos depende do tipo do sistema. Os produtos do pacote têm um nível relativamente elevado de defeitos, ao passo que, geralmente, os sistemas críticos têm uma densidade de defeitos muito menor.

A justificativa para aceitar os defeitos é que, caso o sistema falhe, quando isso acontecer, será menos custoso pagar as consequências da falha do que seria descobrir e remover os defeitos antes da entrega do sistema. No entanto, como discutido no Capítulo 11, a decisão de liberar o software defeituoso não é simplesmente econômica. A aceitabilidade social e política da falha de sistema também deve ser levada em consideração.

Muitos sistemas críticos, como sistemas de aeronaves, sistemas médicos e sistemas de contabilidade, são usados em domínios regulamentados, como transporte aéreo, medicina e finanças. Os governos nacionais definem as regras que se aplicam nesses domínios e nomeiam um órgão regulador para garantir que as empresas sigam estas regras. Na prática, isso significa que os reguladores frequentemente precisam ser convencidos de que os sistemas críticos de software podem ser confiáveis, o que requer uma clara evidência que mostre que esses sistemas são confiáveis.

Portanto, o processo de desenvolvimento de sistemas críticos não está preocupado apenas com a produção de um sistema confiável, mas também deve produzir evidências capazes de convencer os reguladores de que o sistema é confiável. A produção dessas evidências consome grande parte dos custos de desenvolvimento de sistemas críticos e, dessa forma, é um fator importante que contribui para os elevados custos de sistemas críticos. No Capítulo 15, discuto os problemas de se produzirem casos de segurança e confiança.

Figura 13.1 O aumento dos custos de remoção de defeitos residuais

Custo por defeito detectado

Muitos Poucos Muito poucos

Número de defeitos residuais

13.1 Redundância e diversidade

A redundância e a diversidade são estratégias fundamentais para se melhorar a confiança de qualquer tipo de sistema. A redundância significa que a capacidade de reposição está incluída em um sistema que pode ser usado se parte do sistema falhar. A diversidade significa que os componentes redundantes do sistema são de tipos diferentes, aumentando assim as chances de eles não falharem exatamente da mesma maneira.

Usamos redundância e diversidade para melhorar a confiança em nosso cotidiano. Como um exemplo de redundância, a maioria das pessoas mantém lâmpadas sobressalentes em seus lares, para que possam se recuperar rapidamente da falha de uma lâmpada em uso. Comumente, para proteger nossas casas, usamos mais de uma fechadura (redundância) e, geralmente, as fechaduras usadas são de tipos diferentes (diversidade). Isso significa que, mesmo que um intruso descubra uma maneira de abrir uma das fechaduras, ele precisará encontrar uma maneira de abrir a outra fechadura antes que consiga entrar. Como rotina, todos nós devemos fazer *backup* de nossos computadores, mantendo, portanto, cópias redundantes dos dados. Para evitar problemas com falhas de disco, *backups* devem ser mantidos separados do dispositivo externo.

Os sistemas de software projetados para confiança podem incluir componentes redundantes que proporcionem a mesma funcionalidade de outros componentes de sistema. Esses são chaveados no sistema caso o componente principal falhe. Se esses componentes redundantes são diversificados (ou seja, componentes diferentes), um defeito comum em componentes replicados não resultará em uma falha de sistema. A redundância também pode ser fornecida por meio da inclusão de códigos de verificação adicionais, os quais não são estritamente necessários para o funcionamento do sistema. Esse código pode detectar alguns tipos de defeitos antes que falhas sejam causadas. Ele pode invocar mecanismos de recuperação para garantir que o sistema continue a funcionar.

Nos sistemas para os quais a disponibilidade é um requisito essencial, normalmente são usados servidores redundantes. Quando um servidor designado falha, eles entram em funcionamento automaticamente. Às vezes, para garantir que os ataques contra o sistema não possam explorar vulnerabilidades comuns, esses servidores podem ser de diferentes tipos e executar diferentes sistemas operacionais. O uso de diferentes sistemas operacionais é um exemplo de diversidade e redundância de software, em que a funcionalidade similar é fornecida de diferentes maneiras. Na Seção 13.3.4, discuto a diversidade de software mais detalhadamente.

A diversidade e a redundância também podem ser usadas para se ter processos seguros, garantindo que as atividades de processo, como validação de software, não dependam de um único processo ou método. Isso melhora a confiança de software, pois reduz as chances de falhas de processo em que os erros humanos durante o processo de desenvolvimento do software geram erros de software. Por exemplo, as atividades de validação podem incluir testes de programas, inspeções manuais de programa e análise estática como técnica de detecção de defeitos. Essas são técnicas complementares, já que nenhuma técnica pode encontrar os defeitos perdidos por outros métodos. Além disso, diferentes membros de uma equipe podem ser responsáveis por atividades do mesmo processo (por exemplo, uma inspeção de programa). As pessoas lidam com tarefas de maneiras diferentes, dependendo de sua personalidade, experiência e educação, de modo que esse tipo de redundância fornece uma perspectiva diferente do sistema.

Como discuto na Seção 13.3.4, alcançar a diversidade de software não é simples. A diversidade e a redundância tornam os sistemas mais complexos e, geralmente, mais difíceis de serem compreendidos. Não apenas existem mais códigos para se escrever e verificar, mas a funcionalidade adicional também deve ser acrescida ao sistema para detectar falha de componentes e chavear o controle para componentes alternativos. Tal complexidade significa que é mais provável que os programadores cometam erros e menos provável que as pessoas que verificam o sistema encontrem esses erros.

Consequentemente, algumas pessoas pensam que é melhor evitar a redundância e a diversidade de software. Para essas pessoas, a melhor abordagem é projetar o software para ser o mais simples possível, com procedimentos de verificação e validação extremamente rigorosos (PARNAS et al., 1990). Pode-se gastar mais com verificação e validação em virtude da economia que elas proporcionam, pois tornam desnecessário o desenvolvimento de componentes de software redundantes.

Ambas as abordagens são usadas em sistemas críticos de segurança comerciais. Por exemplo, o hardware e o software de controle de voo do Airbus 340 são tão diversos quanto redundantes (STOREY, 1996). O software de controle de voo no Boeing 777 tem como base um hardware redundante, mas cada computador executa o mesmo software, que foi extensivamente validado. O sistema de controle de voo do Boeing 777 centra-se na

simplicidade em vez da redundância. Ambos os aviões são muito confiáveis, de modo que as duas abordagens de confiança, a simples e a diversa, podem ser bem-sucedidas.

13.2 Processos confiáveis

Processos confiáveis de software são processos projetados para produzir softwares confiáveis. A empresa que usa um processo confiável pode ter certeza de que o processo foi devidamente aprovado e documentado, e que as técnicas de desenvolvimento apropriadas foram utilizadas para desenvolvimento de sistemas críticos. A justificativa para investir em processos confiáveis é que um bom processo de software conduzirá a um software com menos erros e, portanto, menores probabilidades de falhar na execução. A Tabela 13.1 mostra alguns dos atributos dos processos confiáveis de software.

Muitas vezes, as evidências de que um processo confiável foi usado são importantes para convencer um regulador de que práticas eficazes da engenharia de software foram aplicadas no desenvolvimento de software. Geralmente, os desenvolvedores de sistema apresentam um modelo do processo a um regulador, junto com a evidência de que o processo foi seguido. O regulador também precisa ser convencido de que o processo é usado de forma consistente por todos os participantes do processo, e que ele pode ser usado em diferentes projetos de desenvolvimento, o que significa que o processo deve ser explicitamente definido e passível de ser repetido.

1. Um processo definido explicitamente é aquele que tem um modelo definido para conduzir o processo de produção de software. Durante o processo, deve ocorrer a coleta de dados para demonstrar que todos os passos do modelo de processo foram executados.
2. Um processo repetitivo é aquele que não depende da interpretação e do julgamento individual. Pelo contrário, pode ser repetido em diversos projetos, por diferentes equipes, independentemente de quem esteja envolvido no desenvolvimento. Isso é particularmente importante para sistemas críticos, com longos ciclos de desenvolvimento durante os quais, muitas vezes, ocorrem mudanças significativas na equipe de desenvolvimento.

Os processos confiáveis fazem uso da redundância e da diversidade para alcançar a confiabilidade. Geralmente, eles incluem diversas atividades com o mesmo objetivo. Por exemplo, as inspeções e testes têm o objetivo de descobrir erros em um programa. As abordagens são complementares, para que, juntas, possam descobrir uma maior quantidade de erros do que com uma única técnica.

As atividades feitas em processos confiáveis certamente dependem do tipo de software que está sendo desenvolvido. No entanto, em geral, essas atividades devem ser orientadas a evitar a introdução de erros em um sistema, detectando e removendo erros e mantendo atualizadas as informações sobre os processos em si. Exemplos de atividades que podem integrar um processo confiável incluem:

1. Revisões de requisitos, para verificar se estes são, na medida do possível, completos e consistentes.
2. Gerenciamento de requisitos, para assegurar que as mudanças nos requisitos sejam controladas e que o impacto das possíveis mudanças nos requisitos seja compreendido por todos os desenvolvedores afetados pela mudança.

Tabela 13.1 Atributos dos processos confiáveis

Característica de processo	Descrição
Documentável	O processo deve ter um modelo de processo definido que estabelece as atividades do processo e a documentação que deve ser produzida durante essas atividades.
Padronizado	Um amplo conjunto de padrões de desenvolvimento de software que abrange a produção e a documentação de software deve estar disponível.
Auditável	O processo deve ser compreensível para pessoas diferentes daquelas que participam do processo, as quais podem verificar se os padrões de processo estão sendo seguidos e fazer sugestões de melhorias ao processo.
Diverso	O processo deve incluir atividades de verificação e validação redundantes e diversas.
Robusto	O processo deve ser capaz de se recuperar de falhas de atividades individuais de processos.

3. Especificação formal, em que um modelo matemático do software é criado e analisado. Eu discuto os benefícios da especificação formal no Capítulo 12. Talvez o benefício mais importante seja que ela força uma análise muito detalhada dos requisitos de sistema. É provável que essa análise descubra problemas de requisitos ignorados durante as revisões.
4. Sistema de modelagem, em que o projeto de software é expressamente documentado como um conjunto de modelos gráficos, e as ligações entre os requisitos e esses modelos são explicitamente documentados.
5. Inspeções de projeto e programa, em que as diferentes descrições do sistema são inspecionadas e controladas por pessoas diferentes. Muitas vezes, as inspeções são conduzidas por *checklists* de erros comuns de projeto e de programação.
6. Análise estática, em que as verificações automatizadas são realizadas no código-fonte do programa. Estas buscam as anomalias capazes de indicar omissões ou erros de programação. A análise estática é discutida no Capítulo 15.
7. Planejamento e gerenciamento de testes, em que um abrangente conjunto de testes de sistemas é projetado. O processo de teste deve ser gerenciado com cuidado para demonstrar que esses testes fornecem a cobertura dos requisitos de sistema e que foram corretamente aplicados no processo de testagem.

Assim como as atividades do processo que se centram no desenvolvimento e testes de sistema, os processos de gerenciamento de qualidade e de gerenciamento de mudanças também devem ser bem definidos. Embora as atividades específicas em um processo confiável possam variar de uma empresa para outra, a necessidade de qualidade efetiva e de gerenciamento de mudanças é universal.

Os processos de gerenciamento de qualidade (discutidos no Capítulo 24) estabelecem um conjunto de padrões de processo e produto. Estes também incluem atividades que capturam informações sobre os processos para demonstrar que tais padrões foram seguidos. Por exemplo, pode haver um padrão definido para a realização de inspeções de programa; o líder da equipe de inspeção é responsável por documentar o processo para demonstrar que o padrão de inspeção tem sido seguido.

O gerenciamento de mudanças, discutido no Capítulo 25, está relacionado com o gerenciamento de mudanças em um sistema, garantindo que as mudanças aceitas sejam efetivamente implementadas, e confirmando que os próximos *releases* de software incluam as mudanças planejadas. Um problema comum com o software ocorre quando componentes errados são incluídos em uma construção de sistema. Isso pode ocasionar uma situação na qual uma execução de sistema inclua componentes não verificados durante o processo de desenvolvimento. Para garantir que isso não ocorra, os procedimentos de gerenciamento de configuração devem ser definidos como parte do processo de gerenciamento de mudanças.

Como discutido no Capítulo 3, existe a opinião generalizada de que as abordagens ágeis não são realmente adequadas para processos confiáveis (BOEHM, 2002). As abordagens ágeis centram-se no desenvolvimento de software, e não em documentar o que foi feito. Normalmente, elas têm uma abordagem bastante informal para o gerenciamento de mudanças e de qualidade. As abordagens baseadas em planos costumam ser preferidas para o desenvolvimento de sistemas confiáveis, que criam a documentação para que as autoridades reguladoras e outros *stakeholders* externos do sistema possam compreender. No entanto, os benefícios das abordagens ágeis são igualmente aplicáveis a sistemas críticos. Existem relatos de sucesso na aplicação de métodos ágeis nesse domínio (LINDVALL et al., 2004), e é provável que se desenvolvam variações adequadas dos métodos ágeis para a engenharia de sistemas críticos.

13.3 Arquiteturas de sistemas confiáveis

Como já discutido, o desenvolvimento de sistemas confiáveis deve basear-se em um processo confiável. No entanto, embora provavelmente seja necessário um processo confiável para criar sistemas confiáveis, esse, por si só, não é suficiente para garantir a confiança do sistema. Você também precisa projetar uma arquitetura de sistemas de segurança, especialmente quando a tolerância a defeitos é necessária. Isso significa que a arquitetura deve ser projetada para incluir componentes e mecanismos redundantes que permitam que o controle seja comutado de um componente para outro.

Exemplos de sistemas que podem precisar de arquiteturas tolerantes a defeitos são: sistemas de aeronaves que devem estar em funcionamento durante todo o período de voo, sistemas de telecomunicações e sistemas críticos de comando e controle. Pullum (2001) descreve os diferentes tipos de arquitetura tolerantes a defeitos que foram propostos, e Torres-Pomales (2000), pesquisa técnicas de tolerância a defeitos de software.

A realização mais simples de uma arquitetura confiável está em servidores replicados, em que dois ou mais servidores efetuam a mesma tarefa. Os pedidos de processamento são canalizados por meio de um componente de gerenciamento de servidor que encaminha cada solicitação para um servidor em particular. Esse componente também mantém o acompanhamento das respostas de servidor. Em caso de falha de servidor, que normalmente é detectada por falta de respostas, o servidor defeituoso é comutado para fora do sistema. Os pedidos não processados são reenviados a outros servidores para processamento.

Essa abordagem de servidor replicado é amplamente usada em sistemas de processamento de transações, em que é fácil manter cópias das transações a serem processadas. Sistemas de processamento de transações são projetados para que os dados sejam atualizados uma vez que uma transação for concluída corretamente, de modo que atrasos no processamento não afetam a integridade do sistema. Essa pode ser uma maneira eficiente de se usar o hardware se o servidor de *backup* for aquele que geralmente é usado para tarefas de baixa prioridade. Se houver um problema com um servidor primário, seu processamento é transferido para o servidor de *backup*, que dá alta prioridade a esse trabalho.

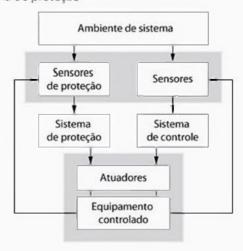
Embora os servidores replicados forneçam a redundância, eles não costumam fornecer a diversidade. Geralmente, o hardware é idêntico e executa a mesma versão do software. Portanto, eles podem lidar com falhas de hardware e de software localizadas em uma única máquina. Eles não podem lidar com problemas de projeto de software que causam falhas em todas as versões de software ao mesmo tempo. Como já discutido na Seção 13.1, para lidar com falhas de projetos de software, um sistema precisa incluir diversos softwares e hardwares.

A diversidade e a redundância de software podem ser implementadas em diferentes estilos de arquitetura. No restante desta seção serão descritos alguns deles.

13.3.1 Sistemas de proteção

Um sistema de proteção é um sistema especializado, associado a algum outro sistema. Geralmente, é um sistema de controle de algum processo, como um processo de fabricação de produtos químicos ou um sistema de controle de equipamentos, como o sistema em um trem sem condutor. Um exemplo de um sistema de proteção pode ser um sistema em um trem que detecta se o trem passou por um sinal vermelho. Caso isso aconteça e não exista uma indicação de que o sistema de controle está desacelerando o trem, então o sistema de proteção automaticamente aciona os freios do trem para levá-lo a uma parada. Os sistemas de proteção monitoram o ambiente de forma independente e, se os sensores indicarem algum problema com o qual o sistema controlado não esteja lidando, então o sistema de proteção é ativado para parar o processo ou o equipamento.

A Figura 13.2 ilustra o relacionamento entre um sistema de proteção e um sistema controlado. O sistema de proteção monitora os equipamentos controlados e o meio ambiente. Se um problema é detectado, ele emite comandos para os atuadores desligarem o sistema ou chamarem outros mecanismos de proteção, como a aber-

Figura 13.2 Arquitetura de sistema de proteção

tura de uma válvula de alívio de pressão. Observe que existem dois conjuntos de sensores. Um conjunto é usado para monitoramento normal de sistema, e o outro, especificamente para o sistema de proteção. Em caso de falha de sensor, existem *backups* que permitirão ao sistema de proteção continuar em operação. Também pode haver atuadores redundantes no sistema.

Um sistema de proteção inclui apenas a funcionalidade crítica necessária para mover o sistema de um estado potencialmente inseguro para um estado seguro (desligamento de sistema). Trata-se de uma instância de uma arquitetura mais geral, tolerante a defeitos em um sistema principal, que é suportada por um sistema menor e mais simples de *backup*, somente com as funções essenciais. Por exemplo, o software norte-americano de controle de veículos espaciais tem um sistema de *backup* que inclui a funcionalidade 'levar você para casa', ou seja, se o sistema de controle principal falhar, o sistema de *backup* pode pousar o veículo.

A vantagem desse tipo de arquitetura é que o software de sistema de proteção pode ser muito mais simples do que o software que está controlando o processo protegido. A única função do sistema de proteção é monitorar a operação e assegurar que o sistema seja levado a um estado seguro no caso de uma emergência. Portanto, é possível investir mais em esforços direcionados a prevenção e detecção de defeitos. Você pode verificar que a especificação de software está correta e consistente e que o software está correto com relação a sua especificação. O objetivo é garantir que a confiabilidade do sistema de proteção seja tal que este tenha uma probabilidade muito baixa de falha sob demanda (por exemplo, 0,001). Dado que, no sistema de proteção, as demandas devem ser raras, uma probabilidade de falha sob demanda de 1/1.000 significa que falhas de sistema de proteção realmente são muito raras.

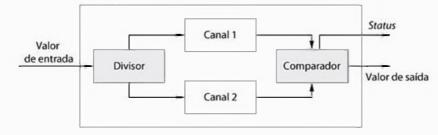
13.3.2 Arquiteturas de automonitoramento

Arquitetura de automonitoramento é uma arquitetura de sistema em que o sistema é projetado para monitorar seu próprio funcionamento e, em caso de detecção de problema, tomar alguma ação de solução. Isso é feito por meio de cálculos em canais separados e pela comparação das saídas desses cálculos. Se as saídas forem idênticas e estiverem disponíveis ao mesmo tempo, então se considera que o sistema está operando corretamente. Se os resultados forem diferentes, então uma falha é assumida. Quando isso ocorre, o sistema costuma gerar uma exceção de falha na linha de *status* de saída, a qual ocasionará a transferência de controle para outro sistema. Essa situação é ilustrada na Figura 13.3.

Para serem eficazes na detecção de defeitos de hardware e software, os sistemas de automonitoramento precisam ser projetados de modo que:

1. O hardware usado em cada canal seja diversificado. Na prática, isso pode significar que cada canal usa um tipo diferente de processador para realizar os cálculos necessários, ou o *chipset* que compõe o sistema pode ser proveniente de fabricantes diferentes. Isso reduz a probabilidade de que defeitos comuns de projeto do processador afetem os cálculos.
2. O software usado em cada canal seja diversificado. Caso contrário, os mesmos erros de software poderiam ocorrer ao mesmo tempo, em cada canal. Na Seção 13.3.4, discuto as dificuldades de se conseguir um software verdadeiramente diversificado.

Por si só, essa arquitetura pode ser usada em situações nas quais seja importante que os cálculos sejam corretos, mas que a disponibilidade não seja essencial. Se as respostas de cada canal são diferentes, o sistema simplesmente desliga. Para muitos sistemas de diagnóstico e tratamento médicos, a confiabilidade é mais importante que a disponibilidade, pois uma resposta incorreta de sistema pode levar o paciente a receber o tratamento incorreto.

Figura 13.3 Arquitetura de automonitoramento

No entanto, se eventualmente um erro leva o sistema simplesmente a se desligar, isso gera um inconveniente, mas o paciente não sai prejudicado.

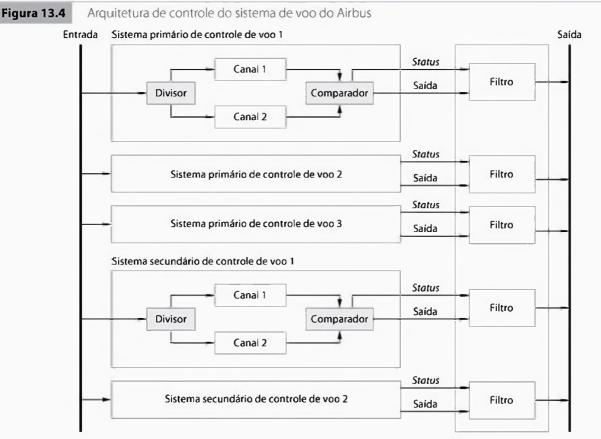
Em situações nas quais a alta disponibilidade é necessária, é preciso usar, em paralelo, vários sistemas de autoverificação. Você precisa de uma unidade de comutação que detecta defeitos e seleciona o resultado de um dos sistemas, em situações em que ambos os canais estejam produzindo uma resposta consistente. Tal abordagem é usada no sistema de controle de voo da série de aeronaves do Airbus 340, em que cinco computadores de autoverificação são usados. A Figura 13.4 é um diagrama simplificado que ilustra essa organização.

No sistema de controle do Airbus, cada um dos computadores de controle de voo realizam os cálculos em paralelo, usando as mesmas entradas. As saídas são conectadas a filtros de hardware que detectam se o *status* indica um defeito e, em caso afirmativo, a saída desse computador é desligada. Em seguida, a saída é obtida de um sistema alternativo. Portanto, é possível que quatro computadores falhem e, mesmo assim, a operação da aeronave continue. Em mais de 15 anos de operação, nunca houve relatos de situações em que o controle da aeronave tenha sido perdido devido à falha total do sistema de controle de voo.

Os projetistas do sistema Airbus tentaram conseguir a diversidade de várias maneiras diferentes:

1. Os computadores primários de controles de voo usam um processador diferente dos sistemas secundários de controle de voo.
2. O *chipset* usado em cada canal dos sistemas primário e secundário é fornecido por um fabricante diferente.
3. O software dos sistemas secundários de controle de voo fornece apenas funcionalidade crítica — é menos complexo que o software primário.
4. O software para cada canal dos sistemas primário e secundário é desenvolvido por equipes diferentes, usando linguagens diferentes de programação.
5. Linguagens de programação diferentes são usadas nos sistemas primário e secundário.

Como é discutido na próxima seção, isso não garante a diversidade, mas reduz a probabilidade de falhas comuns em canais diferentes.

Figura 13.4 Arquitetura de controle do sistema de voo do Airbus

13.3.3 Programação N-*version*

As arquiteturas de automonitoração são exemplos de sistemas em que a programação multiversão é usada para fornecer redundância e diversidade de software. Essa noção de programação multiversão foi derivada de sistemas de hardware em que a noção de redundância modular tripla (TMR, do inglês *triple modular redundancy*) foi usada por muitos anos para construir sistemas tolerantes a falhas de hardware (Figura 13.5).

Em um sistema TMR, a unidade de hardware é replicada três vezes (em alguns casos, mais). A saída de cada unidade é passada para um comparador de saída que, geralmente, é implementado como um sistema de votação. Esse sistema compara todas as entradas e, se duas ou mais forem iguais, então o valor é a saída. Se uma das unidades falhar e não produzir a mesma saída que as outras unidades, essa saída é ignorada. Um gerente de defeitos pode tentar reparar o aparelho defeituoso automaticamente, mas se for impossível, o sistema é automaticamente reconfigurado para colocar a unidade fora de operação. O sistema, então, continua a funcionar com duas unidades.

Essa abordagem de tolerância a defeitos baseia-se no fato de a maioria das falhas de hardware ser resultante de falhas de componentes, e não de defeitos de projeto. Os componentes, por sua vez, tendem a falhar de forma independente. Assume-se que todas as unidades de hardware, quando totalmente operacional, executam conforme a especificação. Portanto, existe uma baixa probabilidade de falha simultânea de componentes em todas as unidades de hardware.

Naturalmente, todos os componentes poderiam ter um defeito comum de projeto e, portanto, todos poderiam produzir a mesma resposta (errada). Usar unidades de hardware com uma especificação comum, mas projetadas e construídas por diferentes fabricantes, reduz as chances de uma falha de modo comum. Supõe-se que é pequena a probabilidade de equipes diferentes cometerem o mesmo erro de projeto ou fabricação.

Uma abordagem semelhante pode ser usada para software tolerante a defeitos, em que diversas versões de um sistema de software executam em paralelo (AVIZIENIS, 1985; AVIZIENIS, 1995). Essa abordagem de tolerância a defeitos de software, ilustrada na Figura 13.6, foi usada em sistemas de sinalização ferroviária, sistemas de aeronaves e sistemas de proteção de reator.

Usando uma especificação comum, o mesmo sistema de software é implementado por várias equipes. Essas versões são executadas em computadores separados. Suas saídas são comparadas a um sistema de votação, e saídas inconsistentes, ou aquelas que não são produzidas em tempo, são rejeitadas. Em caso de uma falha, pelo menos três versões de sistema devem estar disponíveis para que duas versões sejam consistentes.

A programação N-*version* pode ser mais econômica do que arquiteturas de autoverificação em sistemas para os quais é necessário um alto nível de disponibilidade. No entanto, ainda são necessárias várias equipes diferentes para se desenvolverem versões diferentes do software, o que eleva os custos de desenvolvimento de software. Como resultado, essa abordagem é usada somente em sistemas nos quais é impossível fornecer um sistema de proteção que possa proteger contra falhas críticas de segurança.

13.3.4 Diversidade de software

Todas as arquiteturas tolerantes a defeitos citadas anteriormente dependem da diversidade de software para obtenção de tolerância a defeitos. Isso é baseado na suposição de que diversas implementações da mesma especificação (ou, para sistemas de proteção, uma parte da especificação) são independentes. Elas não devem incluir erros comuns e, por isso, não falharão da mesma forma, ao mesmo tempo. Para tanto, o software deve ser escrito

Figura 13.5 Redundância modular tripla

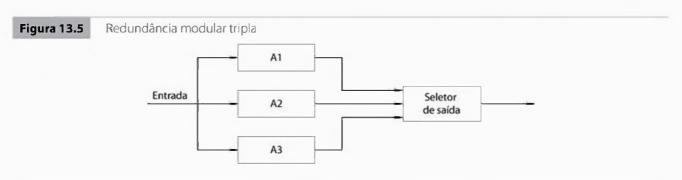
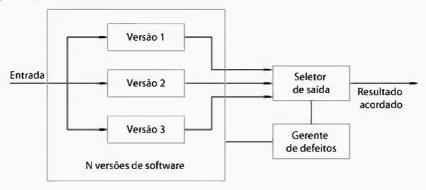



Figura 13.6 Programação N-*version*

por equipes diferentes, as quais não devem se comunicar durante o processo de desenvolvimento, reduzindo, desse modo, as chances de mal-entendidos ou interpretações equivocadas da especificação.

A empresa que está adquirindo o sistema pode incluir políticas explícitas de diversidade que se destinam a maximizar as diferenças entre as versões de sistema. Por exemplo:

1. Ao incluir requisitos de que diferentes métodos de projetos devem ser usados. Por exemplo, uma equipe pode ser necessária para produzir um projeto orientado a objetos, e outra equipe, um projeto orientado a funções.
2. Ao estabelecer que as implementações devem ser escritas em diferentes linguagens de programação. Por exemplo, em um sistema de três versões, Ada, C++ e Java podem ser usadas para escrever as versões de software.
3. Ao exigir o uso de diferentes ferramentas e ambientes de desenvolvimento para o sistema.
4. Ao exigir, explicitamente, que algoritmos diferentes sejam usados em algumas partes da implementação. No entanto, isso limita a liberdade da equipe de projeto e pode ser de difícil conciliação com os requisitos de desempenho de sistema.

Cada equipe de desenvolvimento deve trabalhar com uma especificação detalhada de sistema (às vezes, chamada *V-spec*), derivada da especificação de requisitos de sistema (AVIZIENIS, 1995). Esta deve ser suficientemente detalhada, a fim de garantir que não haja ambiguidades na especificação. Assim como a especificação de funcionalidade do sistema, a especificação detalhada deve definir onde devem ser geradas as saídas de sistema para comparação.

Idealmente, as diversas versões de sistema não devem ter dependências e, dessa forma, devem falhar de maneiras completamente diferentes. Se esse for o caso, a confiabilidade geral de um sistema diversificado é obtida pela multiplicação das confiabilidades de cada canal. Dessa forma, se cada canal tiver uma probabilidade de falha sob demanda de 0,001, então a POFOD global de um sistema de três canais (com todos os canais independentes) será um milhão de vezes maior do que a confiabilidade de um sistema de canal único.

Contudo, na prática, é impossível alcançar a independência completa do canal. Foi demonstrado experimentalmente que as equipes independentes de projeto frequentemente cometem os mesmos erros ou compreendem mal as mesmas partes da especificação (BRILLIANT et al., 1990; KNIGHT e LEVESON, 1986; LEVESON, 1995). Existem várias razões para isso:

1. Os membros de equipes diferentes são frequentemente da mesma origem cultural e podem ter sido educados com a mesma abordagem e mesmos livros didáticos. Isso significa que eles podem encontrar as mesmas dificuldades de compreensão e ter dificuldades comuns de comunicação com especialistas de domínio. É bastante possível que eles possam cometer os mesmos erros e projetem os mesmos algoritmos para solucionar um problema.
2. Se os requisitos forem incorretos ou baseados em mal-entendidos sobre o ambiente do sistema, então os erros se refletirão em cada implementação do sistema.
3. Em um sistema crítico, o *V-spec* é um documento detalhado baseado nos requisitos de sistema, que fornece detalhes completos para as equipes sobre a forma como o sistema deve se comportar. Não pode haver margem para interpretação pelos desenvolvedores de software. Se houver algum erro nesse documento, este será apresentado a todas as equipes de desenvolvimento e implementado em todas as versões do sistema.

Uma maneira de se reduzir a possibilidade de erros comuns de especificação é desenvolver especificações detalhadas para o sistema de maneira independente e definir as especificações em diferentes linguagens. Uma equipe de desenvolvimento pode trabalhar a partir de uma especificação formal, enquanto outra equipe pode partir de um modelo de sistema baseado em estados, e uma terceira, a partir de uma especificação de linguagem natural. Isso ajuda a evitar alguns erros de interpretação de especificação, mas não consegue resolver o problema de erros de especificação. Além disso, introduz a possibilidade de erros na tradução de requisitos, gerando especificações inconsistentes.

Em uma análise de experimentos, Hatton (1997) concluiu que um sistema de três canais era algo entre cinco a nove vezes mais confiável do que um sistema de um único canal. Ele concluiu que as melhorias na confiabilidade que poderiam ser obtidas mediante a atribuição de mais recursos para uma única versão poderiam não se igualar a isso; e, dessa forma, as abordagens N-*version* são suscetíveis a obter sistemas mais confiáveis do que as abordagens de versão única.

No entanto, o que não está claro é se as melhorias na confiabilidade de um sistema multiversão valem os custos extras de desenvolvimento. Para muitos sistemas, os custos adicionais podem não ser justificáveis, assim como uma única versão de sistema, bem projetada, pode ser boa o suficiente. É somente em sistemas críticos de segurança e de missão crítica, em que os custos de falha são muito elevados, que o software multiversão pode ser necessário. Mesmo nessas situações (por exemplo, um sistema de aeronave), pode ser suficiente fornecer um *backup* simples, com funcionalidade limitada, até que o sistema principal possa ser reparado e reiniciado.

13.4 Programação confiável

Neste livro, evito discussões sobre programação, porque é quase impossível discutir esse tema sem entrar em detalhes acerca de uma linguagem de programação específica. Atualmente, existem tantas linguagens e abordagens diferentes no desenvolvimento de software que tenho evitado usar uma linguagem única para os exemplos neste livro. No entanto, quando se considera a engenharia da confiança, existe um conjunto de boas práticas de programação universalmente aceitas, que ajudam a reduzir os defeitos nos sistemas entregues.

Uma lista de diretrizes de boas práticas é mostrada no Quadro 13.1. Elas podem ser aplicadas em qualquer linguagem de programação e usadas para o desenvolvimento de sistemas, embora a forma como os sistemas são usados dependa de linguagens e anotações específicas usadas em seu desenvolvimento.

Diretriz 1: Limitar a visibilidade de informação em um programa

Um princípio de proteção que é adotado por organizações militares é a 'necessidade de conhecer'. As informações são repassadas apenas aos indivíduos que precisam conhecer essa informação para poderem desempenhar suas funções. As informações que não são diretamente relevantes para seu trabalho são retidas.

Ao programar, você deve adotar um princípio semelhante para controlar o acesso a variáveis e estruturas de dados que usa. Os componentes de programa devem ter acesso apenas aos dados que necessitam para sua implementação. Outros dados de programa devem ser inacessíveis e, portanto, ocultados. Se você oculta informa-

Quadro 13.1 Diretrizes de boas práticas para programação confiável

Diretrizes de programação confiável
1. Limitar a visibilidade de informação em um programa
2. Verificar todas as entradas para validade
3. Fornecer um tratador para todas as exceções
4. Minimizar o uso de construções propensas a erros
5. Fornecer recursos de reiniciação
6. Verificar limites de vetor
7. Incluir *timeouts* ao chamar componentes externos
8. Nomear todas as constantes que representam valores do mundo real

ções, estas não podem ser corrompidas pelos componentes de programa que não devem usá-las. Se a interface permanece a mesma, a representação de dados pode ser alterada sem afetar os outros componentes de sistema.

Isso pode ser alcançado por meio da implementação de estruturas de dados em seu programa, como tipos abstratos de dados. Um tipo abstrato de dado é um tipo de dado no qual a estrutura interna e a representação de uma variável desse tipo estão ocultas. A estrutura e os atributos do tipo não são visíveis externamente, e todo acesso aos dados é feito por meio de operações. Por exemplo, você pode ter um tipo abstrato de dado que representa uma fila de pedidos de serviço. As operações devem incluir *get* e *put*, que acrescentam e removem itens da fila, e uma operação que retorna o número de itens na fila. Inicialmente, você pode implementar a fila como um vetor, mas posteriormente pode decidir alterar a implementação para uma lista ligada. Isso pode ser conseguido sem qualquer alteração de código, usando a fila, pois a representação de fila nunca é acessada diretamente.

Você também pode usar tipos abstratos de dados para a implementação de verificações de que o valor atribuído esteja dentro do intervalo permitido. Por exemplo, digamos que você deseja representar a temperatura de um processo químico, no qual as temperaturas permitidas estão no intervalo de 20° a 200° Celsius. Ao incluir uma verificação no valor a ser atribuído no âmbito da operação de tipo abstrato de dado, você pode garantir que o valor da temperatura nunca esteja fora do intervalo requerido.

Em algumas linguagens orientadas a objeto, você pode implementar tipos abstratos de dados usando definições de interface, em que você declara a interface para um objeto sem fazer referência a sua implementação. Por exemplo, você pode definir uma interface Fila, que suporta os métodos para colocar objetos na fila, removê-los da fila e consultar o tamanho da fila. Na classe de objeto que implementa essa interface, os atributos e métodos devem ser privados para essa classe.

Diretriz 2: Verificar todas as entradas para validade

Todos os programas tomam entradas de seu ambiente e os processam. A especificação faz suposições sobre essas entradas que refletem seu uso no mundo real. Por exemplo, pode-se supor que um número de conta bancária seja sempre um inteiro positivo de oito dígitos. Em muitos casos, no entanto, a especificação de sistema não define que ações devem ser tomadas se a entrada estiver incorreta. Inevitavelmente, os usuários cometem erros e, às vezes, introduzem dados errados. Às vezes, como discutido no Capítulo 14, os ataques maliciosos em um sistema dependem da introdução deliberada de uma entrada incorreta. Mesmo quando as entradas vêm de sensores ou de outros sistemas, esses sistemas podem estar errados e fornecer valores incorretos.

Você sempre deve verificar a validade das entradas logo que elas são lidas, a partir do ambiente do programa operacional. As verificações envolvidas certamente dependem das entradas em si, mas há outras possíveis verificações, como as que apresento a seguir.

1. *Verificações de intervalos*. Você pode esperar que as entradas estejam dentro de determinado intervalo. Por exemplo, uma entrada que representa uma probabilidade deve estar dentro do intervalo 0,0 a 1,0; uma entrada que representa a temperatura da água em estado líquido deve estar entre 0° e 100° Celsius, e assim por diante.
2. *Verificações de tamanhos*. Você pode esperar que as entradas sejam de um determinado número de caracteres (por exemplo, oito caracteres para representar uma conta bancária). Em outros casos, o tamanho pode não ser fixo, mas pode haver um limite superior realista. Por exemplo, é improvável que o nome de uma pessoa tenha mais de 40 caracteres.
3. *Verificações de representações*. Você pode esperar que uma entrada seja de um tipo específico, representada em uma forma-padrão. Por exemplo, nomes de pessoas não incluem caracteres numéricos, bem como endereços de e-mail são compostos de duas partes, separadas por um sinal de @ etc.
4. *Verificação de razoabilidade*. Quando a entrada é uma de uma série e você sabe algo sobre os relacionamentos entre os membros dessa série, então você pode verificar se o valor da entrada é razoável. Por exemplo, se o valor da entrada representa as leituras de um medidor residencial de energia elétrica, então você pode esperar que a quantidade de eletricidade usada seja aproximadamente a mesma no período correspondente do ano anterior. Naturalmente haverá variações, mas as diferenças de ordem da magnitude sugerem a existência ou não de um problema.

As ações a serem tomadas, se uma verificação de validação de entrada falhar, não dependem do tipo de sistema a ser implementado. Em alguns casos, deve-se relatar o problema para o usuário e solicitar que o valor seja reinserido. Quando o valor vem de um sensor, você deve usar o valor válido mais recente. Em sistemas embutidos de tempo real, para que o sistema possa continuar em operação, você pode precisar estimar o valor baseado no histórico.

Diretriz 3: Fornecer um tratador para todas as exceções

Durante a execução de programa, é inevitável a ocorrência de erros ou eventos inesperados. Estes podem surgir devido a um defeito de programa ou podem resultar de circunstâncias externas imprevisíveis. Um erro ou um evento inesperado durante a execução de um programa é chamado 'exceção'. Exemplos de exceções podem ser uma falha de energia de sistema, uma tentativa de acesso a dados inexistentes ou, ainda, um *overflow* ou *underflow* numérico.

As exceções podem ser causadas pelas condições de hardware ou software. Quando ocorre uma exceção, ela deve ser gerenciada pelo sistema. Isso pode ser feito dentro do próprio programa ou pode envolver a transferência de controle para um mecanismo de tratamento de exceção de sistema. Normalmente, o mecanismo de gerenciamento de exceções do sistema relata o erro e encerra a execução. Portanto, para garantir que as exceções de programa não causem falhas de sistema, você deve definir um tratador de exceções para todas as exceções que possam ocorrer, e certificar-se de que todas as exceções sejam detectadas e tratadas explicitamente.

Em linguagens de programação como C, as declarações *if* devem ser usadas para detectar exceções e transferir o controle para o código de tratamento de exceção. Isso significa que é necessário verificar exceções de maneira explícita, em qualquer parte do programa que elas possam ocorrer. No entanto, essa abordagem aumenta a complexidade da tarefa de tratamento de exceções, aumentando as chances de erros e, portanto, tratando incorretamente a exceção.

Algumas linguagens de programação, como Java, C++ e Ada, incluem construções de suporte ao tratamento de exceções, para que não sejam necessárias instruções extras para verificação de exceções. Essas linguagens de programação incluem um tipo especial interno (muitas vezes, chamado Exception), e diferentes exceções podem ser declaradas desse tipo. Quando ocorre uma situação excepcional, a exceção é sinalizada e o sistema de linguagem transfere em tempo de execução o controle para um tratador de exceções. Essa é uma seção de código que define os nomes de cada exceção e as medidas apropriadas para lidar com cada exceção (Figura 13.7). Observe que o tratador de exceções está fora do fluxo normal de controle, e que esse fluxo não continua após a exceção ter sido tratada.

Tratadores de exceção costumam fazer uma ou mais ações entre as três adiante:

1. Sinalizar para um componente de nível superior que ocorreu uma exceção e fornecer informações para esse componente sobre o tipo de exceção. Você pode usar essa abordagem quando um componente chama outro e o componente que está chamando precisa saber se o componente chamado foi executado com êxito. Se não, cabe ao componente chamado tomar medidas para se recuperar do problema.
2. Realizar algum processamento alternativo diferente do que estava inicialmente previsto. Portanto, o tratador de exceções toma algumas ações para se recuperar do problema. O processamento pode continuar normalmente ou o tratador de exceções pode indicar que ocorreu uma exceção para que um componente que está chamando fique ciente do problema.
3. Passar o controle para um sistema de *run-time* de suporte que lida com a exceção. Geralmente, esse é o *default* para quando os defeitos ocorrem em um programa (por exemplo, quando ocorre um *overflow* de valor numérico). A ação usual do sistema de *run-time* é parar o processamento. Você só deve usar essa abordagem quando for possível mover o sistema para um estado seguro e quieto, antes de entregar o controle para o sistema de *run-time*.

Figura 13.7 Tratador de exceções

Tratar as exceções dentro de um programa torna possível detectar e recuperar alguns erros de entradas e eventos externos inesperados. Dessa forma, fornece-se um grau de tolerância a defeitos — o programa detecta os defeitos e pode tomar medidas para se recuperar deles. Como a maioria dos erros de entrada e eventos externos inesperados geralmente é transitória, muitas vezes é possível continuar normalmente a operação após a exceção ter sido processada.

Diretriz 4: Minimizar o uso de construções propensas a erros

Geralmente, os defeitos nos programas e, portanto, muitas falhas de programa, são consequência de erros humanos. Os programadores cometem erros porque perdem o controle de diversos relacionamentos entre as variáveis de estado. Eles escrevem declarações de programas que resultam em mudanças de comportamentos e de estado de sistema inesperadas. As pessoas sempre cometerão erros. No final da década de 1960, contudo, tornou-se evidente que algumas abordagens para programação eram mais propensas a induzir erros em um programa que outras.

Algumas construções de linguagem de programação e técnicas de programação são inerentemente sujeitas a erros e por isso devem ser totalmente evitadas ou, pelo menos, usadas o mínimo possível. Construções potencialmente propensas a erros incluem:

1. *Declarações de desvio incondicional* (go-to). Os perigos das declarações *go-to* foram reconhecidos há muito tempo, em 1968 (DIJKSTRA, 1968), e, como consequência, estas foram excluídas das linguagens de programação modernas. No entanto, elas ainda são permitidas em linguagens como C. O uso de declarações *go-to* conduzem ao 'código espaguete', que é confuso e difícil de compreender e depurar.
2. *Números de ponto flutuante.* A representação de números de ponto flutuante em uma palavra de memória de comprimento fixo é inerentemente imprecisa. É particularmente problemática quando os números são comparados, pois a imprecisão na representação pode conduzir a comparações inválidas. Por exemplo, 3,00000000 pode, algumas vezes, ser representado como 2,99999999 e, às vezes, como 3,00000001. Uma comparação mostraria que esses números são desiguais. Geralmente, os números de ponto fixo, em que um número é representado por determinado número de casas decimais, são mais seguros, pois as comparações exatas são possíveis.
3. *Ponteiros.* As linguagens de programação como C e C++ suportam construções de baixo nível chamadas de ponteiros, que possuem endereços que se referem diretamente às áreas de memória da máquina (que apontam para um local da memória). Erros no uso de ponteiros podem ser devastadores quando definidos incorretamente e, portanto, apontam para a área errada da memória. Eles também criam vínculos de verificação de vetores e outras estruturas mais difíceis de se implementar.
4. *Alocação dinâmica de memória.* A memória de programa pode ser alocada em tempo de execução, e não em tempo de compilação. O perigo disso é que a memória não pode ser adequadamente desalocada; assim, eventualmente, o sistema fica sem memória disponível. Esse pode ser um erro muito difícil de detectar, pois o sistema pode continuar funcionando normalmente por um longo período antes de ocorrer o problema.
5. *Paralelismo.* Quando os processos estão executando de maneira concorrente, pode haver, entre eles, sutis dependências de *timing*. Em geral, os problemas de *timing* não podem ser detectados por inspeção de programa, e a peculiar combinação das circunstâncias que causa problemas de *timing* pode não ocorrer durante os testes de sistema. O paralelismo pode ser inevitável, mas seu uso deve ser cuidadosamente controlado para minimizar as dependências entre processos.
6. *Recursão.* Quando um procedimento ou método se chama ou chama outro procedimento, que, por sua vez, chama o procedimento que o chamou, isso é 'recursão'. O uso da recursão pode resultar em programas concisos, mas pode ser difícil seguir a lógica dos programas recursivos. Portanto, os erros de programação são mais difíceis de serem detectados. Os erros de recursão podem resultar na alocação de toda a memória do sistema enquanto variáveis temporárias de pilha são criadas.
7. *Interrupções.* São meios de forçar o controle de transferência para uma seção de código, independentemente do código em execução no momento. Os perigos são óbvios — a interrupção pode causar o encerramento de uma operação crítica.
8. *Herança.* O problema com a herança na programação orientada a objetos é que o código associado com um objeto não está todo em um só lugar, o que torna mais difícil compreender o comportamento do objeto. Por isso, é mais provável que os erros de programação sejam ignorados. Além disso, a herança, quando combinada com ligações dinâmicas, pode causar problemas de *timing* em tempo de execução. Diferentes instâncias de

um método podem ser vinculadas a uma chamada, dependendo dos tipos de parâmetros. Consequentemente, diferentes quantidades de tempo serão despendidas na busca pela instância do método correto.

9. *Construções do tipo 'alias'*. Ocorre quando, em um programa, mais de um nome é usado para se referir à mesma entidade, por exemplo, se dois ponteiros com nomes diferentes apontarem para o mesmo local de memória. É fácil os leitores de programas perderem as declarações que alteram a entidade quando existem vários nomes para se considerar.
10. *Vetores não limitados*. Em linguagens como C, os vetores são maneiras de acessar a memória, e é possível realizar atribuições para além do final de um vetor. O sistema de *run-time* não verifica se as atribuições na verdade se referem aos elementos do vetor. O *overflow* de *buffer*, em que um invasor deliberadamente constrói um programa para escrever na memória para além do final de um *buffer* implementado como um vetor é uma conhecida vulnerabilidade de proteção.
11. *Processamento de entradas* default. Alguns sistemas fornecem um *default* para o processamento de entradas, independentemente da entrada apresentada ao sistema. Essa é uma brecha de proteção que um invasor pode explorar por meio da apresentação do programa com entradas inesperadas, não rejeitadas pelo sistema.

Alguns padrões para o desenvolvimento de sistemas críticos de segurança proíbem completamente o uso dessas construções. No entanto, uma posição tão extrema normalmente não é prática. Todas essas construções e técnicas são úteis, mas devem ser usadas com cuidado. Sempre que possível, seus efeitos potencialmente perigosos devem ser controlados dentro de tipos abstratos de dados ou objetos. Estes agem como '*firewalls*' naturais que limitam os danos no caso de erros.

Diretriz 5: Fornecer recursos de reiniciação

Muitos sistemas de informação organizacionais são baseados em transações curtas, nas quais processar as entradas de usuário leva um tempo relativamente curto. Esses sistemas são projetados de maneira que as alterações ao banco de dados do sistema sejam finalizadas somente após todos os outros processamentos serem concluídos com êxito. Se algo der errado durante o processamento, o banco de dados não é atualizado e, dessa forma, não se torna inconsistente. Virtualmente, todos os sistemas de comércio eletrônico, em que o cliente só se compromete com a compra na tela final, trabalham dessa forma.

As interações de usuário com sistemas de comércio eletrônico geralmente duram poucos minutos e envolvem processamento mínimo. As transações de banco de dados são curtas e, geralmente, concluídas em menos de um segundo. No entanto, outros tipos de sistemas, como sistemas CAD e sistemas de processamento de texto, envolvem transações longas. Em um sistema de transação longa, o período entre começar a usar o sistema e terminar o trabalho pode ser de vários minutos ou horas. Se o sistema falha durante uma transação longa, todo o trabalho pode ser perdido. Da mesma forma, em sistemas de computação intensivos, como alguns sistemas de ciência eletrônica, minutos ou horas de tratamento podem ser necessários para a conclusão dos cálculos. Todo esse tempo é perdido em caso de falha de sistema.

A todos esses tipos de sistemas, você deve atribuir a capacidade de reinício, que se baseia em manter cópias dos dados coletados ou gerados durante o processamento. O recurso de reinício de sistema deve permitir que ele reinicie com essas cópias, em vez de ter de começar tudo de novo desde o início. Às vezes, essas cópias são chamadas *checkpoints*. Por exemplo:

1. Em um sistema de comércio eletrônico, você pode manter cópias de formulários preenchidos pelo usuário e permitir-lhes acesso e apresentar esses formulários sem que seja necessário preenchê-los novamente.
2. Em um sistema de transação longa ou de computação intensiva, você pode salvar automaticamente os dados a intervalos de poucos minutos e, no caso de uma falha de sistema, reiniciar com os dados salvos mais recentemente. Você também deve permitir erros aos usuários e fornecer-lhes uma maneira de voltar ao *checkpoint* mais recente e recomeçar a partir daí.

Caso ocorra uma exceção, sendo impossível continuar normalmente a operação, você pode tratá-la usando a recuperação de erros anterior. Isso significa que você reinicia o estado do sistema ao estado salvo no *checkpoint* e reinicia também a operação a partir desse ponto.

Diretriz 6: Verificar limites de vetor

Todas as linguagens de programação permitem a especificação de vetores — estruturas de dados sequenciais acessadas por meio de um índice numérico. Geralmente, esses vetores são estabelecidos em áreas contíguas dentro da memória de trabalho de um programa. Os vetores são especificados para serem de um tamanho particular, que reflete como essas áreas trabalham. Por exemplo, se você deseja representar as idades de até dez mil pessoas, você pode declarar um vetor com dez mil posições para armazenar os dados de idade.

Algumas linguagens de programação, como Java, sempre verificam se, quando um valor é inserido em um vetor, o índice está dentro desse vetor. Assim, se um vetor A é indexado de 0 a 10.000, uma tentativa de inserir valores em elementos A [-5] ou A [12345] conduzirá ao surgimento de uma exceção. No entanto, linguagens de programação como C e C++ não incluem automaticamente a verificação de limites de vetores e simplesmente calculam um deslocamento a partir do início do vetor. Portanto, A [12345] teria acesso à palavra que está a 12.345 posições a partir do início do vetor, independentemente de este fazer parte do vetor ou não.

A razão pela qual essas linguagens não incluem a verificação automática de limites de vetores é que esta introduz um *overhead* cada vez que o vetor é acessado. A maioria dos acessos ao vetor está correta, então a verificação de limites é praticamente desnecessária e aumenta o tempo de execução do programa. No entanto, a falta de verificação de limites leva a vulnerabilidades de proteção, como o *overflow* de *buffer*, discutido no Capítulo 14. Mais frequentemente, ela introduz uma vulnerabilidade no sistema, que pode resultar em falha. Se você estiver usando uma linguagem que não inclui a verificação de limites de vetores, sempre deve incluir um código extra que garanta que o índice do vetor esteja dentro dos limites. Isso é facilmente conseguido por meio da implementação do vetor como um tipo abstrato de dados, como já discutido na Diretriz 1.

Diretriz 7: Incluir *timeouts* ao chamar componentes externos

Em sistemas distribuídos, seus componentes executam em computadores diferentes e as chamadas são feitas através da rede, de componente para componente. Para receber algum serviço, o componente A pode chamar o componente B. A espera a resposta de B antes de prosseguir com a execução. No entanto, se por alguma razão o componente B não responder, o componente A não poderá continuar. Ele simplesmente esperará indefinidamente por uma resposta. Uma pessoa que esteja aguardando uma resposta do sistema vê uma falha silenciosa, sem resposta do sistema. Os componentes não têm alternativa senão encerrar o processo em espera e reiniciar o sistema.

Para evitar isso, você sempre deve incluir *timeouts* ao chamar componentes externos. Um *timeout* é uma suposição automática de que um componente chamado falhou e não produzirá uma resposta. Você define um período durante o qual espera receber uma resposta de um componente chamado. Se não receber uma resposta nesse período, você assume que houve uma falha e retoma o controle a partir do componente chamado. Você pode tentar recuperar-se da falha ou dizer ao usuário do sistema o que aconteceu e permitir que ele decida o que fazer.

Diretriz 8: Nomear todas as constantes que representam valores do mundo real

Todos os programas não triviais incluem uma série de valores constantes que representam os valores de entidades do mundo real. Esses valores não são modificados enquanto o programa é executado. Às vezes, eles são constantes absolutas e nunca mudam (por exemplo, a velocidade da luz), mas mais frequentemente são valores que mudam ao longo do tempo de forma relativamente lenta. Por exemplo, um programa para calcular o imposto de pessoas inclui como constantes as taxas de impostos atuais. Estas mudam de um ano para outro, e, portanto, o programa deve ser atualizado com os novos valores constantes.

Você sempre deve incluir uma seção em seu programa na qual você nomeie todos os valores constantes do mundo real que são usados. Ao usar as constantes, você deve se referir a elas pelo nome, e não pelo valor. Na medida em que se relaciona com a confiança, isso tem duas vantagens:

1. A propensão a cometer erros e a usar valores errados é menor. É fácil escrever um número errado e o sistema ficar frequentemente incapaz de detectar um erro. Por exemplo, digamos que uma taxa de imposto é de 34%. Um simples erro de transposição pode levar a sua digitação incorreta, como 43%. No entanto, se você digitar incorretamente um nome (como uma taxa-padrão de imposto), este geralmente será detectado pelo compilador como uma variável não declarada.
2. Quando um valor for alterado, você não precisa buscar em todo o programa para descobrir onde você usou aquele valor. Tudo que você precisa fazer é mudar o valor associado com a declaração de constante. O novo valor será automaticamente incluído em todos os lugares em que ele é necessário.

- Em um programa, a confiança pode ser alcançada evitando-se a introdução de defeitos, detectando e removendo defeitos antes da implantação do sistema e incluindo recursos de tolerância a defeitos, que permitem que o sistema se mantenha em funcionamento depois que um defeito tenha ocasionado uma falha de sistema.
- O uso de redundância e diversidade de hardware, de processos de software e de sistemas de software é essencial para o desenvolvimento de sistemas confiáveis.
- O uso de um processo repetitivo e bem definido é essencial para que os defeitos em um sistema sejam minimizados. O processo deve incluir atividades de verificação e validação em todos os estágios, desde a definição de requisitos até a implementação de sistema.
- Arquiteturas confiáveis de sistemas são arquiteturas projetadas para tolerância a defeitos. Existe uma série de estilos de arquitetura que suportam tolerância a defeitos, incluindo sistemas de proteção, arquiteturas de automonitoramento e programação N-*version*.
- A diversidade de software é difícil de ser obtida, pois é praticamente impossível garantir que cada versão de software seja verdadeiramente independente.
- Programação confiável conta com a inclusão de redundância no programa para verificação da validade de entradas e dos valores das variáveis de programa.
- Algumas técnicas e construções de programação, como as declarações *go-to*, ponteiros, herança, recursão e números de ponto flutuante, são, por natureza, sujeitas a erros. No desenvolvimento de sistemas confiáveis, você deve tentar evitar essas construções.

Software Fault Tolerance Techniques and Implementation. Uma discussão abrangente sobre as técnicas para alcançar softwares de tolerância a defeitos e arquiteturas tolerantes a defeitos. O livro também aborda questões gerais de confiança de software. (PULLUM, L.L. *Software Fault Tolerance Techniques and Implementation*. Artech House, 2001.)

'Software Reliability Engineering: A Roadmap'. Esse artigo desenvolvido por um pesquisador líder em confiabilidade resume o estado da arte em engenharia de confiabilidade de software, além de discutir os desafios de futuras pesquisas. (LYU, M.R. Proc. Future of Software Engineering, *IEEE Computer Society*, 2007.) Disponível em: <http://dx.doi.org/10.1109/FOSE.2007.24>.

13.1 Dê quatro razões pelas quais quase nunca é rentável para empresas garantir que seu software seja livre de defeitos.

13.2 Explique por que é razoável supor que usar processos confiáveis levará à criação de um software confiável.

13.3 Dê dois exemplos de atividades diversas e redundantes que podem ser incorporadas aos processos confiáveis.

13.4 Qual é a característica comum de todos os estilos de arquitetura orientados a suportar a tolerância a defeitos de software?

13.5 Imagine que você está implementando um sistema de controle baseado em software. Sugira circunstâncias em que seria conveniente usar uma arquitetura tolerante a defeitos, e explique por que essa abordagem seria necessária.

13.6 Você é responsável pelo projeto de um comutador de comunicações que tem disponibilidade de 24 horas, sete dias por semana, mas que não é crítico de segurança. Sugira um estilo de arquitetura que possa ser usado para esse sistema. Justifique sua resposta.

13.7 Foi sugerido que um software de controle para uma máquina de radioterapia, usada para tratar pacientes com câncer, deve ser implementado usando programação N-*version*. Você acha que é uma boa sugestão? Comente.

13.8 Dê duas razões pelas quais diferentes versões de um sistema baseado na diversidade de software podem falhar de maneira semelhante.

13.9 Explique por que você deve tratar explicitamente todas as exceções em um sistema planejado para ter um nível de disponibilidade elevado.

13.10 Como discutido neste capítulo, o uso de técnicas para a produção de um software seguro, obviamente inclui custos adicionais significativos. Qual custo extra pode ser justificado se cem vidas forem salvas durante 15 anos de vida útil de um sistema? Será que os mesmos custos podem ser justificados se dez vidas forem salvas? Quanto vale uma vida? Será que as capacidades de ganho salarial das pessoas afetadas fazem diferença para esse julgamento?

REFERÊNCIAS

AVIZIENIS, A. The N-Version Approach to Fault-Tolerant Software. *IEEE Trans. on Software Eng.*, v. SE-11, n. 12, 1985, p. 1491-1501.

AVIZIENIS, A. A. A Methodology of N-Version Programming. In: LYU, M. R. (Org.). *Software Fault Tolerance*. Chichester: John Wiley & Sons, 1995, p. 23-46.

BOEHM, B. Get Ready for Agile Methods, With Care. *IEEE Computer*, v. 35, n. 1, 2002, p. 64-69.

BRILLIANT, S. S.; KNIGHT, J. C.; LEVESON, N. G. Analysis of Faults in an N-Version Software Experiment. *IEEE Trans. On Software Engineering*, v. 16, n. 2, 1990, p. 238-247.

DIJKSTRA, E. W. Go-to statement considered harmful. *Comm. ACM*, v. 11, n. 3, 1968, p. 147-148.

HATTON, L. N-version design *versus* one good version. *IEEE Software*, v. 14, n. 6, 1997, p. 71-76.

KNIGHT, J. C.; LEVESON, N. G. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. on Software Engineering*, v. SE-12, n. 1, 1986, p. 96-109.

LEVESON, N. G. *Safeware*: System Safety and Computers. Reading, Mass.: Addison-Wesley, 1995.

LINDVALL, M.; MUTHIG, D.; DAGNINO, A.; WALLIN, C.; STUPPERICH, M.; KIEFER, D., MAY, D; KAHKONENT. Agile Software Development in Large Organizations. *IEEE Computer*, v. 37, n. 12, 2004, p. 26-34.

PARNAS, D. L.; VAN SCHOUWEN, J.; SHU, P. K. Evaluation of Safety-Critical Software. *Comm. ACM*, v. 33, n. 6, 1990, p. 636-651.

PULLUM, L. L. *Software Fault Tolerance Techniques and Implementation*. Norwood, Mass.: Artech House, 2001.

STOREY, N. *Safety-Critical Computer Systems*. Harlow, Reino Unido: Addison-Wesley, 1996.

TORRES-POMALES, W. *Software Fault Tolerance*: A Tutorial. Disponível em: <http://ntrs.nasa.gov/archive/nasa/casi./20000120144_2000175863.pdf>.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 **14** 15 16 17 18 19 20 21 22 23 24 25 26

Engenharia de proteção

Objetivos

O objetivo deste capítulo é introduzir questões que devem ser consideradas quando você está projetando sistemas de aplicação protegidos. Com a leitura deste capítulo, você:

- entenderá a diferença entre proteção de aplicativos e proteção de infraestrutura;
- saberá como a avaliação de riscos de ciclo de vida e a avaliação de riscos operacionais são usadas para compreender questões de proteção que afetam um projeto de sistema;
- compreenderá arquiteturas de software e as diretrizes de projeto para o desenvolvimento de sistemas protegidos;
- compreenderá o conceito de sobrevivência e por que a análise de sobrevivência é importante para sistemas complexos de software.

Conteúdo

O uso generalizado da Internet na década de 1990 introduziu um novo desafio para engenheiros de software: projetar e implementar sistemas protegidos. Como cada vez mais sistemas foram conectados à Internet, criou-se uma variedade de ataques externos para ameaçar esses sistemas. Os problemas de produção de sistemas confiáveis foram aumentando com o tempo. Os engenheiros de sistemas têm de considerar tanto ameaças de ataques mal-intencionados e tecnicamente qualificados quanto problemas resultantes de erros acidentais no processo de desenvolvimento.

Atualmente, é essencial projetar sistemas para resistir a ataques externos e para recuperar-se desses ataques. Sem precauções de proteção, é quase inevitável que invasores comprometerão os sistemas em rede. Eles podem abusar do hardware do sistema, roubar dados confidenciais ou interromper os serviços oferecidos pelo sistema. A engenharia de proteção de sistema, portanto, é um aspecto cada vez mais importante do processo de engenharia de sistemas.

A engenharia de proteção está preocupada com o desenvolvimento e a evolução de sistemas que possam resistir a ataques mal-intencionados para danificar o sistema ou seus dados. A engenharia de software de proteção é parte do campo mais geral da proteção de computadores. Isso se tornou prioridade para as empresas e indivíduos, uma vez que é crescente o número de criminosos que tentam explorar sistemas em rede para fins ilegais. Os engenheiros de software devem estar cientes das ameaças à proteção enfrentadas pelos sistemas e das maneiras que essas ameaças podem ser neutralizadas.

Minha intenção neste capítulo é introduzir o tema engenharia de proteção para engenheiros de software, com foco em questões de projeto que afetam a proteção da aplicação. O capítulo não é sobre a proteção do computador como um todo e, portanto, não cobre tópicos como criptografia, controle de acesso, mecanismos de autorização, vírus, cavalos de Troia etc. Esses são descritos em detalhes em outros textos sobre proteção de computador (ANDERSON, 2008; BISHOP, 2005; PFLEEGER e PFLEEGER, 2007).

Este capítulo acrescenta novos pontos à discussão sobre proteção já existente neste livro. Você deve ler este material junto com:

- Seção 10.1, em que explico como proteção e confiança estão intimamente relacionadas;
- Seção 10.4, em que apresento a terminologia de proteção;
- Seção 12.1, em que apresento a noção geral de especificação dirigida a riscos;
- Seção 12.4, em que discuto questões gerais de especificação de requisitos de proteção;
- Seção 15.3, em que explico uma série de abordagens para testes de proteção.

Quando você considerar questões de proteção, deve considerar o software de aplicação (o sistema de controle, sistema de informação etc.) e a infraestrutura sobre a qual esse sistema é construído (Figura 14.1). A infraestrutura para aplicações complexas pode incluir:

- uma plataforma de sistema operacional, como Linux ou Windows;
- outras aplicações genéricas que são executadas no sistema, como *browsers* de Web e clientes de e-mail;
- um sistema de gerenciamento de banco de dados;
- *middleware* que ofereça suporte à computação distribuída e acesso ao banco de dados;
- bibliotecas de componentes reusáveis usados pelo software de aplicação.

A maioria dos ataques externos foca as infraestruturas de sistemas porque os componentes de infraestrutura (por exemplo, *browsers* de Web) são bem conhecidos e amplamente disponíveis. Os invasores podem investigar esses sistemas em busca de pontos fracos e compartilhar as informações sobre vulnerabilidades que descobrirem. Como muitas pessoas usam o mesmo software, ataques têm ampla aplicabilidade. Vulnerabilidades de infraestrutura podem levar invasores a obter acesso não autorizado a um sistema de aplicação e seus dados.

Na prática, há uma distinção importante entre proteção de aplicação e proteção de infraestrutura:

1. Proteção de aplicação é um problema de engenharia de software em que os engenheiros devem assegurar que o sistema é projetado para resistir a ataques.
2. Proteção de infraestrutura é um problema de gerenciamento em que os gerentes de sistema configuram a infraestrutura para resistir a ataques. Gerentes de sistema precisam configurar a infraestrutura para fazer o uso mais eficaz de quaisquer recursos de proteção da infraestrutura disponíveis. Eles também precisam corrigir vulnerabilidades de proteção de infraestrutura que vêm à luz quando o software é usado.

Gerenciamento de proteção de sistemas não é uma tarefa única; de fato, inclui uma gama de atividades, como gerenciamento de usuários e permissões, implantação e manutenção de software de sistema, monitoração, detecção e recuperação de ataques:

1. Gerenciamento de usuários e de permissões inclui adicionar e remover usuários de sistema, garantindo que mecanismos apropriados de autenticação de usuários estejam sendo usados e configurando as permissões no sistema para que os usuários só tenham acesso aos recursos de que necessitem.
2. Implantação e manutenção de sistema de software incluem a instalação de software de sistema e *middleware* e sua correta configuração para que as vulnerabilidades de proteção sejam evitadas. Envolve também a atualização regular desse software com novas versões ou *patches*, que reparem problemas de proteção descobertos.

Figura 14.1 Camadas de sistema em que a proteção pode ser comprometida

Aplicação
Componentes reusáveis e bibliotecas
Middleware
Gerenciamento de banco de dados
Aplicações genéricas compartilhadas (*browsers*, e-mail etc.)
Sistema operacional

3. Monitoração, detecção e recuperação de ataques incluem atividades que monitorem o sistema para acesso não autorizado, detecção e execução de estratégias para resistir a ataques e atividades de *backup* para que a operação normal possa ser reiniciada após um ataque externo.

Gerenciamento de proteção é de vital importância, mas geralmente não é considerado parte da engenharia de proteção de aplicações. Em vez disso, a engenharia de proteção de aplicações está preocupada com o projeto de um sistema para que ele seja tão protegido quanto possível, dadas as restrições de orçamento e usabilidade. Parte desse processo é o 'projeto por gerenciamento', em que são projetados sistemas que minimizem a possibilidade de erros de gerenciamento de proteção que levem a ataques bem-sucedidos.

Para sistemas críticos de controle e sistemas embutidos, é normal selecionar uma infraestrutura adequada para suportar o sistema de aplicação. Por exemplo, os desenvolvedores de sistemas embutidos costumam escolher um sistema operacional de tempo real que forneça à aplicação embutida os recursos de que ele precisa. Requisitos de segurança e vulnerabilidades conhecidos podem ser levados em conta. Isso significa que uma abordagem holística pode ser adotada para engenharia de proteção. Requisitos de proteção de aplicações podem ser implementados através da infraestrutura ou da própria aplicação.

No entanto, sistemas de aplicações em uma organização geralmente são implementados usando-se a infraestrutura existente (sistema operacional, banco de dados etc.). Portanto, os riscos do uso dessa infraestrutura e seus recursos de proteção devem ser levados em conta como parte do processo de projeto de sistema.

14.1 Gerenciamento de riscos de proteção

O gerenciamento e a avaliação de riscos de proteção são essenciais para a eficácia da engenharia de proteção. O gerenciamento de riscos se preocupa com possíveis perdas que possam resultar de ataques a ativos do sistema e com um balanço dessas perdas em relação aos custos de procedimentos de proteção que possam reduzi-las. As empresas de cartão de crédito fazem isso o tempo todo. É relativamente fácil introduzir novas tecnologias para reduzir a fraude de cartão de crédito. No entanto, muitas vezes é mais barato para eles compensar os usuários por suas perdas devidas à fraude do que comprar e implantar a tecnologia de redução de fraudes. Como os custos caem e os ataques aumentam, esse equilíbrio pode mudar. Por exemplo, empresas de cartão de crédito agora estão codificando informações no chip do cartão em vez de usar a fita magnética. Isso torna a cópia do cartão muito mais difícil.

O gerenciamento de riscos é um problema de negócios, e não um problema técnico. Portanto, engenheiros de software não deveriam decidir quais controles devem ser incluídos no sistema. Cabe à gerência sênior decidir se deve ou não aceitar o custo da proteção ou da exposição que resulta de uma falta de procedimentos de proteção. Em vez disso, o papel de engenheiros de software é fornecer orientação técnica e julgamento sobre questões de proteção. Eles são, portanto, essenciais no processo de gerenciamento de riscos.

Como expliquei no Capítulo 12, uma entrada crítica para o processo de avaliação e gerenciamento de riscos é a política de proteção da organização. Essa política aplica-se a todos os sistemas e deve definir o que deve e não deve ser permitido. As políticas de proteção definem condições que sempre devem ser mantidas por um sistema de proteção e, assim, ajuda a identificar os riscos e as ameaças que possam surgir. A política de proteção, portanto, define o que é e o que não é permitido. No processo de engenharia de proteção, você projeta os mecanismos para implementar essa política.

A avaliação de riscos começa antes da decisão de adquirir o sistema. Deve continuar durante todo o processo de desenvolvimento de sistema e depois que o sistema seja colocado em uso (ALBERTS e DOROFEE, 2002). No Capítulo 12, apresentei a ideia de que essa avaliação de riscos é um processo em estágios:

1. *Avaliação preliminar de riscos.* Nesse estágio, as decisões sobre os requisitos detalhados de sistema, o projeto de sistema ou a tecnologia de implementação não foram feitos. O objetivo desse processo de avaliação é decidir se pode ser atingido um nível adequado de proteção a um custo razoável. Se esse for o caso, você pode derivar, em seguida, requisitos de proteção específicos para o sistema. Você não tem informações sobre vulnerabilidades potenciais no sistema ou os controles que são incluídos em componentes de sistema reusados ou *middleware*.

2. *Avaliação de riscos de ciclo de vida.* Essa avaliação de riscos ocorre durante o ciclo de vida de desenvolvimento de sistema e é informada pelas decisões de projeto e implementação de sistemas técnicos Os resultados da avaliação podem gerar alterações dos requisitos de proteção e à adição de novos requisitos. As vulnerabilida-

des conhecidas e potenciais são identificadas e esse conhecimento é usado para informar a tomada de decisão sobre a funcionalidade de sistema e como ela é implementada, testada e entregue.

3. *Avaliação de riscos operacionais.* Após um sistema ser implantado e colocado em uso, a avaliação de riscos deve continuar a levar em consideração como o sistema é usado, assim como as propostas de novos requisitos e alterações. Suposições sobre os requisitos operacionais de quando o sistema foi especificado podem estar incorretas. As mudanças organizacionais podem significar que o sistema será usado de maneiras diferentes do que foi originalmente planejado. A avaliação de riscos operacionais, portanto, estimula requisitos de proteção que devem ser implementados conforme o sistema evolui.

A avaliação preliminar de riscos concentra-se em derivar requisitos de proteção. No Capítulo 12, mostro como um conjunto inicial de requisitos de proteção pode ser especificado a partir de uma avaliação preliminar de riscos. Nesta seção, concentro-me no ciclo de vida e avaliação de riscos operacionais para ilustrar como a especificação e o projeto de um sistema são influenciados pela tecnologia e pela maneira como o sistema é usado.

Para realizar uma avaliação de riscos, você precisa identificar possíveis ameaças a um sistema. Uma maneira de fazer isso é por meio do desenvolvimento de um conjunto de 'casos de mau uso' (ALEXANDER, 2003; SINDRE e OPDAHL, 2005). Já discuti como os casos de uso — típicas interações com um sistema — podem ser usados para especificar requisitos de sistema. Casos de mau uso são cenários que representam interações mal-intencionadas com um sistema. Você pode usá-los para discutir e identificar possíveis ameaças e, consequentemente, determinar requisitos de proteção de sistema. Eles podem ser usados junto com casos de uso quando da derivação de requisitos de sistema.

Pfleeger e Pfleeger (2007) caracterizam as ameaças em quatro tópicos, que podem ser o ponto de partida para a identificação de possíveis casos de mau uso. São os seguintes:

1. Ameaças de interceptação que possam permitir a um invasor obter acesso a um ativo. Assim, um possível caso de mau uso para o MHC-PMS pode ser uma situação na qual um invasor obtém acesso aos registros individuais de um paciente bem conhecido.
2. Ameaças de interrupção que permitem a um invasor deixar parte do sistema indisponível. Portanto, um caso de mau uso pode ser um ataque de negação de serviço em um servidor de banco de dados de sistema.
3. Ameaças de modificação que permitem a um invasor violar um ativo de sistema. No MHC-PMS, isso poderia ser representado por um caso de mau uso, em que um invasor altera as informações em um registro de paciente.
4. Ameaças de fabricação que permitem a um invasor inserir informações falsas em um sistema. Isso talvez não seja uma ameaça crível no MHC-PMS, mas certamente seria uma ameaça em um sistema bancário, em que transações falsas podem ser adicionadas ao sistema que transfere dinheiro para a conta bancária do criminoso.

Os casos de mau uso são úteis não apenas na avaliação preliminar de riscos, mas podem ser usados para análise de proteção em análise de riscos de ciclo de vida e operacionais. Eles fornecem uma base útil para se simular ataques hipotéticos ao sistema e se avaliar as implicações de proteção das decisões de projeto que foram feitas.

14.1.1 Avaliação de riscos de ciclo de vida

Com base em políticas de proteção da organização, uma avaliação preliminar de riscos deve identificar os requisitos de proteção mais importantes de um sistema. Estes refletem como a política de proteção deve ser implementada nessa aplicação, identificam os ativos a serem protegidos e decidem qual abordagem deve ser usada para oferecer essa proteção. No entanto, manutenção de proteção é dar a atenção aos detalhes. É impossível que os requisitos iniciais de proteção tenham considerado todos os detalhes que afetam a proteção.

A avaliação de riscos de ciclo de vida identifica os detalhes de projeto e de implementações que afetam a proteção. Essa é a importante distinção entre a avaliação de riscos de ciclo de vida e a avaliação preliminar de riscos. A avaliação de riscos de ciclo de vida afeta a interpretação dos requisitos de proteção existentes, gera novos requisitos e influencia o projeto geral do sistema.

Ao avaliar os riscos, nesse estágio, você deve ter informações mais detalhadas sobre o que deve ser protegido e conhecer alguma coisa sobre as vulnerabilidades do sistema. Algumas dessas vulnerabilidades serão inerentes às escolhas de projeto. Por exemplo, uma vulnerabilidade em todos os sistemas baseados em senhas ocorre quando um usuário autorizado revela sua senha para um usuário não autorizado. Como alternativa, se uma empresa tem uma política de desenvolvimento de software em C, você saberá que a aplicação pode ter vulnerabilidades porque a linguagem não inclui verificação de limites de vetor.

A avaliação de riscos de proteção deve ser parte de todas as atividades do ciclo de vida, desde a engenharia de requisitos até a implantação do sistema. O processo é semelhante ao processo de avaliação preliminar de riscos, com a adição de atividades interessadas na identificação e avaliação de vulnerabilidades de projeto. O resultado da avaliação de riscos é um conjunto de decisões de engenharia que afetam o projeto ou a implementação de sistema ou limitam a maneira como este é usado.

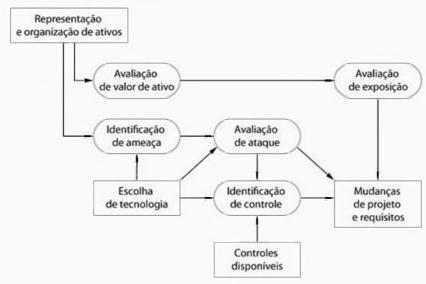
Um modelo do processo de análise de riscos de ciclo de vida, baseado no processo de análise preliminar de riscos que descrevi na Figura 12.4, é mostrado na Figura 14.2. A diferença mais importante entre esses processos é que agora você tem informações sobre a representação e distribuição de informações e sobre a organização do banco de dados para os ativos de alto nível que precisam ser protegidos. Você também está ciente das decisões de projeto mais importantes, como o software a ser reusado, controles e proteção de infraestrutura etc. Com base nessas informações, sua análise identifica as alterações nos requisitos de proteção e no projeto de sistema, para fornecer proteção adicional aos ativos importantes de sistema.

Dois exemplos ilustram como requisitos de proteção são influenciados por decisões sobre a representação e distribuição de informações:

1. Você pode tomar uma decisão de projeto para separar informações sobre tratamentos recebidos e informações pessoais de pacientes, com uma chave para vincular esses registros. As informações sobre o tratamento são muito menos sensíveis do que as informações pessoais de pacientes, e, portanto, podem não precisar de uma proteção maior. Se a chave estiver protegida, um invasor só será capaz de acessar informações de rotina e não será capaz de ligar essa informação a um paciente específico.
2. Suponha que, no início de uma sessão, é tomada uma decisão de projeto de copiar os registros de pacientes para um sistema local de cliente. Isso permite que o trabalho continue se o servidor não estiver disponível e torna possível para um funcionário do sistema de saúde acessar os registros de pacientes de um *laptop*, mesmo que nenhuma conexão de rede esteja disponível. No entanto, agora você tem dois conjuntos de registros para proteger, e as cópias de clientes estão sujeitas a riscos adicionais, como o roubo dos computadores portáteis. Logo, é necessário decidir quais controles devem ser usados para obter a redução de riscos. Por exemplo, os registros de cliente nos *laptops* podem precisar ser criptografados.

Para ilustrar como as decisões de tecnologias de desenvolvimento influenciam a proteção, suponha que um provedor de saúde decidiu construir um MHC-PMS usando um sistema de informação disponível no mercado, para a manutenção de registros de pacientes. Esse sistema precisa ser configurado para cada tipo de clínica em que será usado. Essa decisão foi tomada porque esse sistema parece oferecer funcionalidade mais ampla pelo menor custo de desenvolvimento, bem como tempo de implantação mais rápido.

Quando você desenvolver uma aplicação com o reúso de um sistema existente, precisa aceitar as decisões de projeto feitas pelos desenvolvedores do sistema. Vamos supor que algumas dessas decisões de projeto sejam as seguintes:

Figura 14.2 Análise de riscos de ciclo de vida

1. Os usuários são autenticados por uma combinação de nome e senha de *login*. Nenhum outro método de autenticação é suportado.
2. A arquitetura de sistema é cliente-servidor, e os clientes têm acesso a dados por meio de um *browser*-padrão em um PC cliente.
3. A informação é apresentada aos usuários como um formulário Web editável. Os usuários podem alterar as informações no local e carregar as informações revisadas no servidor.

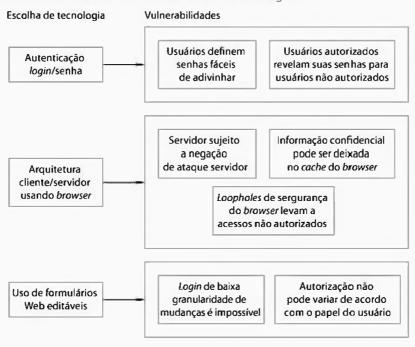
Para um sistema genérico, essas decisões de projeto são perfeitamente aceitáveis, mas uma análise de riscos de ciclo de vida revela que elas têm vulnerabilidades associadas. Exemplos de possíveis vulnerabilidades são mostrados na Figura 14.3.

Uma vez identificadas as vulnerabilidades, é necessário decidir que passos seguir para reduzir os riscos associados. Muitas vezes, isso implicará decisões sobre os requisitos de proteção de sistema adicionais ou sobre o processo operacional de uso do sistema. Não tenho espaço para discutir todos os requisitos que podem ser propostos para resolver as vulnerabilidades inerentes, mas alguns exemplos de requisitos podem ser os seguintes:

1. Um programa verificador de senhas deve ser disponibilizado e executado diariamente. As senhas de usuário que aparecem no dicionário do sistema devem ser identificadas, e usuários com senhas fracas devem ser relatados para os administradores de sistema.
2. O acesso ao sistema deve ser autorizado apenas em computadores clientes que tenham sido aprovados e registrados pelos administradores do sistema.
3. Todos os computadores clientes devem ter um único *browser* instalado, aprovado pelos administradores.

Se um sistema de prateleira for usado, não será possível incluir um verificador de senhas na aplicação de sistema propriamente dita; portanto, deve-se usar um sistema separado. Os verificadores de senhas analisam a força das senhas de usuários quando estas são criadas e notificam os usuários caso eles escolham senhas fracas. Portanto, senhas vulneráveis podem ser identificadas de modo razoável e rápido, logo após terem sido criadas, e medidas podem ser tomadas para garantir que os usuários alterem sua senha.

O segundo e terceiro requisitos implicam todos os usuários sempre acessarem o sistema por meio do mesmo *browser*. Você pode decidir qual é o *browser* mais protegido quando o sistema for implantado e instalá-lo em todos os computadores clientes. Atualizações de proteção são simplificadas porque não é necessário atualizar *browsers* diferentes quando vulnerabilidades de proteção são descobertas e corrigidas.

Figura 14.3 As vulnerabilidades associadas com escolhas de tecnologias

14.1.2 Avaliação de riscos operacionais

A avaliação de riscos de proteção deve continuar durante toda a vida do sistema, a fim de identificar riscos emergentes e alterações que possam ser necessárias para lidar com esses riscos. Esse processo é chamado avaliação de riscos operacionais. Novos riscos podem surgir devido à alteração de requisitos de sistema, a alterações na infraestrutura de sistema ou no ambiente em que o sistema é usado.

O processo de avaliação de riscos operacionais é semelhante ao processo de avaliação de riscos de ciclo de vida, mas com a adição de mais informações sobre o ambiente em que o sistema é usado. O ambiente é importante porque suas características podem gerar novos riscos para o sistema. Por exemplo, um sistema está sendo usado em um ambiente no qual os usuários são frequentemente interrompidos. Um possível risco é que, ao ser interrompido, o usuário precise deixar seu computador sozinho. Logo, pode acontecer de uma pessoa não autorizada acessar as informações no sistema. Isso poderia gerar um requisito para a proteção de tela, protegida por senha, a ser executado após certo período de inatividade.

14.2 Projeto para proteção

Geralmente, é verdade que é difícil adicionar proteção a um sistema depois que ele tenha sido implementado. Portanto, durante o processo de projeto de sistemas, você precisa levar em consideração questões de proteção. Nesta seção, eu foco principalmente em questões de projeto de sistema, porque esse tópico não tem tido a atenção que merece em livros sobre proteção do computador. Erros e questões de implementação também têm grande impacto sobre a proteção, mas esses frequentemente são dependentes de tecnologia específica usada. Recomendo o livro de Viega e McGraw (2002) como uma boa introdução à programação para proteção.

Concentro-me em um número de questões gerais, independentes de aplicações relevantes para o projeto de sistemas protegidos:

1. *Projeto de arquitetura* — como as decisões de projeto de arquitetura afetam a proteção de um sistema?
2. *Boas práticas* — quais são as boas práticas aceitáveis para o projeto de sistemas protegidos?
3. *Projeto para implantação* — qual suporte deve ser projetado em sistemas para evitar a introdução de vulnerabilidades quando um sistema for implantado para uso?

Naturalmente, essas questões não são importantes somente para a proteção. Cada aplicação é diferente, e o projeto de proteção também precisa considerar a finalidade, a importância e o ambiente operacional da aplicação. Por exemplo, se você estiver projetando um sistema militar, precisará adotar o modelo de classificação de proteção (secreto, ultrassecreto etc.). Se você estiver projetando um sistema que mantém informações pessoais, poderá ter de levar em consideração uma legislação de proteção de dados que coloca restrições em como os dados são gerenciados.

Existe uma estreita relação entre confiança e proteção. O uso de redundância e diversidade, fundamental para atingir a confiança, pode significar que um sistema é capaz de resistir e recuperar-se de ataques a caraterísticas do projeto ou implementação específicas. Mecanismos de suporte a um alto nível de disponibilidade podem ajudar o sistema a se recuperar dos chamados ataques de negação de serviço, em que o objetivo de um invasor é derrubar o sistema e interromper seu funcionamento adequado.

O projeto de um sistema de proteção inevitavelmente envolve compromissos. Em um sistema, é possível projetar várias medidas de proteção que reduzirão as chances de um ataque bem-sucedido. No entanto, muitas vezes as medidas de proteção exigem uma grande quantidade adicional e, assim, afetam o desempenho geral de um sistema. Por exemplo, você pode reduzir as chances de informações confidenciais serem divulgadas criptografando-as. No entanto, isso significa que os usuários da informação precisam esperar para ser descriptografados, o que pode retardar seu trabalho.

Também existem tensões entre proteção e usabilidade. Em alguns casos, as medidas de proteção exigem que o usuário lembre e forneça informações adicionais (por exemplo, várias senhas). No entanto, os usuários, às vezes, esquecem essas informações, e, dessa forma, proteção adicional significa que eles não poderão usar o sistema. Portanto, os projetistas precisam encontrar um equilíbrio entre a proteção, o desempenho e a usabilidade. Isso vai depender do tipo de sistema e de onde ele será usado. Por exemplo, em um sistema militar, os usuários estão familiarizados com sistemas de alta proteção; assim, estão dispostos a aceitar e seguir processos que requerem verificações frequentes. Contudo, em um sistema de negociação de ações, as interrupções de operações para verificações de proteção seriam totalmente inaceitáveis.

14.2.1 Projeto de arquitetura

Como já discutido no Capítulo 11, a escolha da arquitetura de software pode ter efeitos profundos sobre as propriedades emergentes de um sistema. Se uma arquitetura inadequada é usada, pode ser muito difícil manter a confidencialidade e a integridade das informações do sistema ou garantir um nível desejado de disponibilidade de sistema.

Ao projetar uma arquitetura de sistema que mantenha a proteção, é necessário considerar duas questões fundamentais:

1. *Proteção* — como o sistema deve ser organizado para que os ativos críticos possam ser protegidos contra ataques externos?
2. *Distribuição* — como os ativos de sistema devem ser distribuídos de modo que os efeitos de um ataque bem-sucedido sejam minimizados?

Essas questões são potencialmente conflitantes. Se você colocar todos seus ativos em um lugar, poderá construir camadas de proteção em torno deles. Como será necessário construir apenas um sistema de proteção individual, é possível suportar um sistema forte, com várias camadas de proteção. No entanto, se essa proteção falhar, todos seus ativos estarão comprometidos. Adicionar várias camadas de proteção também afeta a usabilidade de um sistema, de modo que se pode dizer que é mais difícil atender aos requisitos de usabilidade e desempenho de sistema.

Contudo, se você distribuir seus ativos, ficará mais caro protegê-los, pois os sistemas de proteção precisam ser implementados para cada cópia. Dessa maneira, você não pode custear muitas camadas de proteção. As maiores chances são de a proteção ser violada. No entanto, caso isso ocorra, você não sofre uma perda total. Pode ser possível duplicar e distribuir os ativos de informações de modo que, se uma cópia estiver corrompida ou inacessível, a outra cópia poderá ser usada. No entanto, se a informação for confidencial, manter cópias adicionais aumentará o risco de um intruso ter acesso a essa informação.

Para o sistema de registro de pacientes, é adequado o uso de uma arquitetura de banco de dados centralizado. Para fornecer a proteção, você usa uma arquitetura em camadas com os ativos críticos protegidos no nível mais baixo do sistema, com várias camadas de proteção em torno deles. A Figura 14.4 ilustra essa situação para o sistema de registro de pacientes, no qual os ativos críticos a serem protegidos são os registros individuais de pacientes.

Para acessar e modificar os registros de pacientes, um invasor precisa penetrar três camadas de sistema:

1. *Proteção em nível de plataforma*. O nível superior controla o acesso à plataforma, no qual o sistema de registro de pacientes é executado. Isso costuma envolver um usuário identificado em um determinado computador. Geralmente, a plataforma também inclui suporte para a manutenção da integridade de arquivos no sistema, *backups* etc.
2. *Proteção em nível de aplicação*. O próximo nível de proteção é construído dentro da própria aplicação. Envolve um usuário acessando a aplicação, sendo autenticado e obtendo autorização para tomar medidas, como a visualização ou modificação de dados. Apoio ao gerenciamento de integridade de aplicações específicas pode estar disponível.
3. *Proteção em nível de registro*. Esse nível é invocado quando é necessário o acesso a registros específicos e consiste em verificar se um usuário está autorizado a realizar as operações solicitadas sobre esse registro. Nesse nível, a proteção também pode envolver a criptografia para assegurar que os registros não possam ser navegados usando um *browser* de arquivos. A verificação de integridade, com uso, por exemplo, de *checksums* criptográficos, pode detectar alterações feitas fora dos mecanismos normais de atualização de registros.

O número de camadas de proteção necessárias em qualquer aplicação depende da criticidade dos dados. Nem todas as aplicações precisam de proteção em nível de registro e, portanto, o controle de acesso mais geral é mais usado. Para alcançar a proteção, você não deve permitir que as mesmas credenciais de usuário sejam usadas em todos os níveis. Caso você tenha um sistema baseado em senhas, o ideal é que a senha de aplicação seja diferente da senha de sistema e da senha em nível de registro. No entanto, é difícil para os usuários se lembrarem de várias senhas; além disso, os usuários não suportam os repetidos pedidos de autenticação. Portanto, é muitas vezes necessário comprometer a proteção a favor da usabilidade do sistema.

Figura 14.4 Uma arquitetura de proteção em camadas

Proteção em nível de plataforma

Autenticação de sistema | Autorização de sistema | Gerenciamento de integridade de arquivos

Proteção em nível de aplicação

Login de banco de dados | Autorização de banco de dados | Gerenciamento de transações | Recuperação de banco de dados

Proteção em nível de registro

Autorização de acesso de registros | Criptografia de registros | Gerenciamento de integridade de registros

Registros de pacientes

Se a proteção de dados for um requisito essencial, uma arquitetura cliente-servidor deve ser usada, com os mecanismos de proteção incorporados ao servidor. No entanto, se a proteção for comprometida, as perdas associadas ao ataque podem ser altas, assim como os custos de recuperação (por exemplo, todas as credenciais de usuário podem precisar ser reemitidas). O sistema é vulnerável a ataques de negação de serviço, que sobrecarregam o servidor e tornam impossível a qualquer pessoa acessar o banco de dados do sistema.

Caso você considere que os ataques de negação de serviço são um grande risco, você pode decidir usar, para a aplicação, uma arquitetura de objetos distribuídos. Nessa situação, ilustrada na Figura 14.5, os ativos do sistema estão distribuídos entre várias plataformas diferentes, com mecanismos de proteção separados para cada uma dessas plataformas. Um ataque a um nó pode significar que alguns ativos estão indisponíveis, mas ainda assim seria possível fornecer alguns serviços de sistema. Os dados podem ser replicados em todos os nós do sistema, de modo que a recuperação de ataques seja simplificada.

A Figura 14.5 mostra a arquitetura de um sistema bancário para negociação de ações e fundos em mercados de Nova York, Londres, Frankfurt e Hong Kong. O sistema é distribuído para que os dados sobre cada mercado sejam mantidos em separado. Os ativos necessários para apoiar a atividade crítica de negociação de ações (contas de usuários e preços) são replicados e disponibilizados em todos os nós. Se um nó de sistema for atacado e se tornar indisponível, a atividade crítica de negociação de ações pode ser transferida para outro país, e assim, continuar disponível para os usuários.

Já foi discutida a dificuldade de encontrar um equilíbrio entre a proteção e o desempenho de um sistema. Um problema de projeto de sistema protegido é que, em muitos casos, o estilo de arquitetura mais adequado para o cumprimento de requisitos de proteção pode não ser o melhor para satisfazer os requisitos de desempenho. Por exemplo, digamos que uma aplicação tem um requisito absoluto — manter a confidencialidade de um grande banco de dados — e outro requisito de acesso muito rápido a esses dados. Um elevado nível de proteção sugere a necessidade de camadas de proteção, o que significa que deve haver comunicação entre as camadas do sistema. Isso acarreta uma inevitável sobrecarga de desempenho, que reduz a velocidade de acesso aos dados. Se uma arquitetura alternativa for usada, implementar a proteção e a garantia de confidencialidade poderá ser mais difícil e mais custoso. Em uma situação dessas, é necessário discutir os conflitos inerentes com o cliente de sistema e decidir como estes devem ser resolvidos.

Figura 14.5 Ativos distribuídos em um sistema de negociação de ações

Autenticação e autorização
Sistema de negociação de ações de Nova York
Contas de usuários norte-americanos
Contas de usuários internacionais
História de negociações de ações dos EUA
Dados de patrimônio dos EUA
Preços de ações internacionais
Dados do tesouro dos EUA

Autenticação e autorização
Sistema de negociação de ações de Londres
Contas de usuários do Reino Unido
Contas de usuários internacionais
História de negociações de ações do Reino Unido
Dados de patrimônio do Reino Unido
Preços de ações internacionais
Dados do tesouro do Reino Unido

Autenticação e autorização
Sistema de negociação de ações de Frankfurt
Contas de usuários europeus
Contas de usuários internacionais
História de negociações de ações europeias
Dados de patrimônio da Europa
Preços de ações internacionais
Dados do tesouro da Europa

Autenticação e autorização
Sistema de negociação de ações de Hong Kong
Contas de usuários de Hong Kong
Contas de usuários internacionais
História de negociações de ações de Hong Kong
Dados de patrimônio do Reino Unido
Preços de ações internacionais
Dados do tesouro asiático

14.2.2 Diretrizes de projeto

Não existem regras simples e rápidas de como alcançar a proteção do sistema. Diferentes tipos de sistemas requerem medidas técnicas diferentes para se atingir um nível de proteção aceitável para o proprietário. As atitudes e os requisitos de diferentes grupos de usuários afetam profundamente o que é ou não aceitável. Por exemplo, em um banco, os usuários estão mais dispostos a aceitar um maior nível de proteção e, portanto, procedimentos de proteção mais importunos do que, digamos, em uma universidade.

No entanto, existem diretrizes gerais que têm ampla aplicabilidade durante o projeto de soluções de proteção de sistemas, as quais encapsulam boas práticas de projetos para a engenharia de sistemas de proteção. As diretrizes gerais de projeto de proteção, assim como as discutidas, têm dois usos principais:

1. Ajudam a aumentar a consciência para as questões de proteção em uma equipe de engenharia de software. Muitas vezes, os engenheiros de software centram-se em um objetivo de curto prazo — entregar o software funcionando para os clientes. É comum que eles negligenciem as questões de proteção. Conhecer essas diretrizes pode significar que as questões de proteção sejam consideradas quando forem tomadas as decisões a respeito de projeto de software.
2. Elas podem ser usadas como um *checklist* de revisão para ser usado no processo de validação de sistema. A partir das diretrizes de alto nível discutidas aqui, podem ser derivadas questões mais específicas, que exploram como a proteção de um sistema foi projetada em um sistema.

As dez diretrizes de projeto, resumidas no Quadro 14.1, foram derivadas de diferentes fontes (SCHNEIER, 2000; VIEGA e McGRAW, 2002; WHEELER, 2003). Neste livro, centro-me nas diretrizes particularmente aplicáveis para os processos de especificações e projeto de software. Os princípios mais gerais, como 'Proteja o elo mais fraco em um sistema', 'Mantenha-o simples' e 'Evite proteção por meio da obscuridade' também são importantes, mas, diretamente, menos relevantes para o processo de tomada de decisão de engenharia.

Diretriz 1: Basear as decisões de proteção em uma política explícita de proteção

Uma política de proteção é uma declaração de alto nível que define as condições de proteção fundamentais para uma organização. Ela define o 'quê' de proteção, ao invés de como'. Portanto, a política não deve definir os mecanismos a serem usados para fornecer e garantir a proteção. Em princípio, todos os aspectos da política de proteção devem ser refletidos nos requisitos do sistema; na prática, especialmente se um processo rápido de desenvolvimento de aplicações for usado, é improvável que isso aconteça. Portanto, os projetistas devem consultar a política de proteção, uma vez que ela estabelece um *framework* para a tomada e avaliação de decisões de projeto.

Por exemplo, digamos que você está projetando um sistema de controle de acesso para o MHC-PMS. A política de proteção do hospital pode estabelecer que apenas o corpo clínico credenciado poderá modificar os registros eletrônicos de pacientes. Portanto, seu sistema precisa incluir mecanismos para verificar o credenciamento de qualquer tentativa de modificar o sistema e para rejeitar as modificações de pessoas não credenciadas.

O problema que você pode enfrentar é que muitas organizações não têm uma política explícita de proteção de sistema. Pode ser que, ao longo do tempo, as alterações de sistema tenham sido feitas em resposta aos problemas identificados, mas sem nenhum documento de política geral para orientar a evolução de um sistema. Nestas situações, você precisa trabalhar e documentar a política a partir de exemplos e confirmá-la com os gerentes da empresa.

Diretriz 2: Evitar um único ponto de falha

Em qualquer sistema crítico, tentar evitar um único ponto de falha é uma boa prática de projeto, pois significa que uma única falha não deve resultar em falha do sistema global. Em termos de proteção, significa que você não deve confiar em um único mecanismo de garantia de proteção e que deve empregar várias técnicas diferentes. Esse processo, às vezes, é chamado 'defesa em profundidade'.

Por exemplo, se você usar uma senha para autenticar os usuários de um sistema, você também poderá incluir um mecanismo de autenticação do tipo desafio/resposta, em que os usuários tenham perguntas e respostas pré-registradas no sistema. Assim, para ter acesso, após a autenticação de senha, deve-se responder às perguntas corretamente. Para proteger a integridade dos dados em um sistema, você pode manter um log executável de todas as alterações feitas nos dados (veja Diretriz 5). No caso de uma falha, você pode reproduzir o log para recriar o conjunto de dados. Você também pode fazer uma cópia de todos os dados modificados antes de a alteração ser feita.

Diretriz 3: Falhar de maneira protegida

As falhas de sistema são inevitáveis em todos os sistemas, e, da mesma forma que os sistemas críticos de segurança devem sempre ser do tipo falha-segura (*fail-safe*), os sistemas críticos de proteção sempre devem ser do tipo falha-protegida (*fail-secure*). Quando o sistema falha, você não deve usar os procedimentos de contingência, menos protegidos do que o sistema em si. A falha do sistema não deve significar que um invasor pode acessar dados aos quais ele normalmente não teria acesso.

Quadro 14.1 Diretrizes de projeto para a engenharia de sistemas protegidos

Diretrizes de proteção
1. Basear as decisões de proteção em uma política explícita de segurança
2. Evitar um ponto único de falência
3. Falhar de maneira protegida
4. Equilibrar a proteção e a usabilidade
5. Registrar ações de usuários
6. Usar redundância e diversidade para reduzir riscos
7. Validar todas as entradas
8. Compartimentar seus ativos
9. Projetar para implantação
10. Projetar para recuperabilidade

Por exemplo, no sistema de informação de pacientes, eu sugeri um requisito de que os dados de pacientes sejam transferidos para um cliente de sistema no início de uma consulta. Isso acelera o acesso e significa que ele continuará possível mesmo em casos de indisponibilidade do servidor. Normalmente, o servidor deleta os dados no final da consulta. No entanto, se o servidor falhar, existe a possibilidade de as informações serem mantidas pelo cliente. Nessas condições, uma abordagem de falha-protegida envolve criptografar todos os dados de pacientes armazenados no cliente, o que impossibilita que um usuário não autorizado tenha acesso aos dados.

Diretriz 4: Equilibrar a proteção e a usabilidade

Frequentemente, o equilíbrio entre a proteção e a usabilidade é contraditório. Para criar uma proteção do sistema, você precisa introduzir verificações que autorizem os usuários a usarem o sistema e que certifiquem que eles estão em conformidade com as políticas de proteção. Inevitavelmente, essas verificações fazem exigências aos usuários; eles podem ter de se lembrar de nomes e senhas de *login*, usar o sistema apenas em alguns computadores específicos, e assim por diante. Isso significa que os usuários precisam de mais tempo para começar a usar o sistema — e para usá-lo eficazmente. À medida que você adiciona recursos de proteção a um sistema, é inevitável que ele se torne menos usável. Eu recomendo o livro de Cranor e Garfinkel (2005), que discute uma ampla gama de questões gerais de proteção e usabilidade.

Chega-se a um ponto em que é contraproducente acrescentar novos recursos de proteção em detrimento da usabilidade. Por exemplo, se você demandar aos usuários que insiram múltiplas senhas ou que alterem suas senhas para sequências de caracteres com intervalos frequentes impossíveis de serem memorizadas, eles simplesmente anotarão as senhas. Um invasor (especialmente interno) pode ser capaz de encontrar as senhas anotadas e obter acesso ao sistema.

Diretriz 5: Registrar ações de usuários

Se possível, você sempre deve manter um *log* de ações de usuários, o qual deve, pelo menos, registrar quem fez o que, os ativos usados, a hora e a data da ação. Como discutido na Diretriz 2, caso você mantenha isso como uma lista de comandos executáveis, você terá a opção de repetir o *log* para se recuperar de falhas. Naturalmente, você também precisa de ferramentas que permitam analisar o *log* e detectar ações potencialmente anômalas. Essas ferramentas podem varrer o log, encontrar ações anômalas e, assim, ajudar a detectar ataques, além de rastrear como o invasor ganhou acesso ao sistema.

Além de ajudar na recuperação de falhas, um *log* das ações do usuário é útil porque atua como um impedimento para ataques internos. Quando as pessoas sabem que suas ações estão sendo registradas, elas ficam menos propensas a tomar atitudes não autorizadas. E isso é ainda mais eficaz para ataques casuais — como quando uma enfermeira procura registros de pacientes — ou para detectar ataques em que as credenciais do usuário legítimo são roubadas por meio de engenharia social. Evidentemente, esse meio não é infalível, pois intrusos tecnicamente qualificados podem acessar e alterar o log.

Diretriz 6: Usar redundância e diversidade para reduzir o riscos

A redundância significa a manutenção de mais de uma versão de software ou de dados em um sistema. A diversidade, quando aplicada ao software, significa que as diferentes versões não devem ser colocadas na mesma plataforma ou ser implementadas usando-se as mesmas tecnologias. Portanto, uma vulnerabilidade de plataforma ou tecnologia não afetará todas as versões e pode levar a uma falha comum. No Capítulo 13, expliquei como a redundância e a diversidade são mecanismos fundamentais na engenharia de confiança.

Já discuti exemplos de redundância — manter informações de pacientes no servidor e no cliente, em primeiro lugar no sistema de saúde mental e, em seguida, no sistema distribuído de negociação de ações mostrado na Figura 14.5. No sistema de registros de pacientes, você pode usar diversos sistemas operacionais no cliente e no servidor (por exemplo, Linux no servidor e Windows no cliente). Isso garante que um ataque baseado em uma vulnerabilidade de sistema operacional não afete ambos, servidor e cliente. Naturalmente, é necessário negociar esses benefícios pelo aumento de custos de gerenciamento da manutenção de diferentes sistemas operacionais em uma organização.

Diretriz 7: Validar todas as entradas

Um ataque comum em um sistema envolve fornecer entradas inesperadas ao sistema que o levam a se comportar de uma forma não prevista. Isso pode causar uma parada de sistema, que resulta em perdas do serviço, ou

as entradas podem ser compostas de códigos mal-intencionados que são executados pelo sistema. As vulnerabilidades de *overflow* de *buffer*, primeiro demonstradas no *worm* de Internet (SPAFFORD, 1989) e comumente usadas por invasores (BERGHEL, 2001), podem ser disparadas pelo uso de longas sequências de caracteres de entrada. É o chamado 'envenenamento de SQL', no qual um usuário mal-intencionado insere um fragmento de SQL que é interpretado por um servidor. Esse é outro ataque muito comum.

Como expliquei no Capítulo 13, você pode evitar muitos desses problemas se projetar a validação de entradas em seu sistema. Essencialmente, você nunca deve aceitar uma entrada sem aplicar algumas verificações. Como parte dos requisitos, você deve definir as verificações que devem ser aplicadas e usar o conhecimento de entradas para definir tais verificações. Por exemplo, se um sobrenome deve ser inserido, é necessário verificar se não existem espaços embutidos e se o único sinal de pontuação usado é o hífen. Você também pode verificar o número de caracteres de entrada e rejeitar entradas que sejam, obviamente, muito longas. Por exemplo, provavelmente ninguém tem um nome de família com mais de 40 caracteres, e os endereços provavelmente nunca têm mais de cem caracteres. Caso você use menus para apresentar as entradas permitidas, você evitará alguns dos problemas de validação de entradas.

Diretriz 8: Compartimentar seus ativos

Compartimentar significa que você não deve fornecer o acesso 'tudo-ou-nada' às informações em um sistema. Em vez disso, você deve organizar as informações em compartimentos. Os usuários só devem ter acesso às informações de que eles precisam, e não a todas as informações do sistema. Dessa forma, os efeitos de um ataque podem ser contidos. Algumas informações podem ser perdidas ou danificadas, mas é improvável que todas as informações no sistema sejam afetadas.

Por exemplo, no sistema de informações de pacientes, você deve projetar o sistema de modo que em qualquer consultório o corpo clínico normalmente só tenha acesso aos registros de pacientes que tenham consultas ali. Geralmente, o pessoal não deve ter acesso a todos os registros de pacientes no sistema, o que limita as potenciais perdas ocasionadas por ataques maliciosos. Isso também significa que, se um invasor roubar suas credenciais, a quantidade de danos que este pode causar é limitada.

Dito isso, você também deve ter mecanismos que concedam acesso inesperado ao sistema — digamos que um paciente gravemente doente demande tratamento urgente, sem agendamento. Nessas circunstâncias, você pode usar algum mecanismo alternativo de proteção para superar a compartimentação do sistema. Nessas situações, em que a proteção é relaxada para manutenção da disponibilidade de sistema, é essencial que você use um mecanismo de *log* para registrar o uso do sistema. Dessa forma, você pode rastrear os *logs* de qualquer uso não autorizado.

Diretriz 9: Projetar para implantação

Muitos problemas de segurança surgem porque o sistema, quando implantado em seu ambiente operacional, não está configurado corretamente. Portanto, você sempre deve projetar seu sistema de maneira que sejam incluídos recursos para simplificar a implantação no ambiente do cliente e para verificar possíveis erros e omissões de configuração no sistema implantado. Esse é um tópico importante, que abordarei em detalhes na Seção 14.2.3.

Diretriz 10: Projetar para recuperabilidade

Independentemente de quanto esforço seja dispendido na manutenção de sistemas de proteção, você deve sempre projetar seu sistema com base no pressuposto de que podem ocorrer falhas de proteção. Portanto, você deve pensar em como se recuperar de possíveis falhas e restaurar o sistema para um estado operacional protegido. Por exemplo, você pode incluir um sistema de *backup* de autenticação para o caso de a autenticação de senhas ser comprometida.

Outro exemplo: digamos que uma pessoa de fora da clínica, não autorizada, ganhe acesso ao sistema de registros de pacientes; você não sabe como essa pessoa obteve uma combinação válida de *login*/senha. Nesse caso, não basta alterar as credenciais usadas pelo intruso; é preciso reiniciar o sistema de autenticação. Isso é essencial, pois o intruso pode obter acesso a outras senhas de usuários. Portanto, é necessário garantir que todos os usuários autorizados alterem suas senhas e que pessoas não autorizadas não tenham acesso aos mecanismos de alteração de senhas.

Portanto, você precisa projetar seu sistema para negar o acesso a todos, até que todos tenham alterado suas senhas, e autenticar os usuários reais a alterarem suas senhas, assumindo que as senhas escolhidas podem não estar

protegidas. Uma maneira de fazer isso é usar um mecanismo de desafio/resposta, em que os usuários tenham de responder perguntas para as quais eles já tenham respostas pré-registradas. Esse mecanismo só é invocado quando as senhas são alteradas, permitindo a recuperação de ataques com, relativamente, pouca interrupção de usuário.

14.2.3 Projetar para implantação

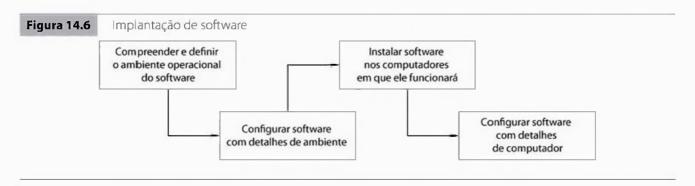
A implantação de um sistema envolve configurar o software para operar em um ambiente operacional, instalar o sistema nos computadores desse ambiente e, em seguida, configurar o sistema instalado para esses computadores (Figura 14.6). A configuração pode ser um processo simples que envolve o estabelecimento de alguns parâmetros internos do software para que reflitam as preferências do usuário. No entanto, às vezes, ela é complexa e exige a definição específica de modelos e regras de negócios que afetam a execução do software.

É nesse estágio do processo que, muitas vezes, as vulnerabilidades são acidentalmente introduzidas no software. Por exemplo, durante a instalação, o software frequentemente precisa ser configurado com uma lista dos usuários permitidos. Quando entregue, essa lista consiste em um *login* de administrador genérico, como 'admin', e a senha *default*, tal como 'senha'. Isso torna mais fácil para um administrador configurar o sistema. Sua primeira ação deve ser introduzir um novo nome de *login* e senha e deletar o nome de *login* genérico. No entanto, é fácil se esquecer de fazer isso. Um invasor que conheça o *login* default pode ser capaz de obter acesso privilegiado ao sistema.

Muitas vezes, a configuração e a implantação são vistas como problemas de administração de sistema e, por isso, são consideradas fora do escopo dos processos de engenharia de software. Certamente, boas práticas de gerenciamento podem evitar muitos problemas de proteção decorrentes de erros de configuração e implantação. No entanto, os projetistas de software têm a responsabilidade de 'projetar para a implantação'. Você sempre deve fornecer suporte interno para implantação, o que reduzirá a probabilidade de os administradores de sistema (ou usuários) cometerem erros ao configurar o software.

Eu recomendo quatro maneiras para incorporar o suporte de implantação em um sistema:

1. *Incluir suporte para visualização e análise de configurações.* Você sempre deve incluir, em um sistema, recursos que permitam aos administradores ou usuários autorizados examinarem a configuração atual do sistema. Surpreendentemente, esse recurso não consta da maioria dos sistemas de software, e os usuários ficam frustrados pela dificuldade em encontrar as definições de configuração. Por exemplo, em uma versão de processador de texto que usei para escrever este capítulo, é impossível ver ou imprimir todas as preferências de sistema em uma única tela. No entanto, se um administrador puder obter uma imagem completa de uma configuração, ficará mais fácil detectar erros e omissões. Idealmente, uma tela de configuração deveria destacar os aspectos potencialmente não seguros da configuração — por exemplo, se uma senha não foi definida.
2. *Minimizar privilégios* default. Você deve projetar softwares em que a configuração *default* de um sistema forneça privilégios essenciais mínimos. Dessa forma, podem ser limitados os possíveis danos ocasionados por intrusos. Por exemplo, a autenticação *default* de administrador de sistema só deve permitir acesso a um programa que permita a um administrador configurar novas credenciais. Este não deve permitir o acesso a quaisquer outros recursos de sistema. Uma vez que as novas credenciais tenham sido definidas, o *login* e a senha *default* devem ser deletados automaticamente.
3. *Localizar definições de configuração.* Ao projetar um suporte de configuração de sistema, você deve garantir que, em uma configuração, tudo o que afeta a mesma parte de um sistema seja configurado no mesmo lugar.

Figura 14.6 Implantação de software

Para usar o exemplo do processador de texto novamente, na versão que eu uso, posso definir algumas informações de proteção, como uma senha para controlar o acesso ao documento, usando o menu de Preferências/Proteção. Outras informações são definidas no menu Ferramentas/Proteger Documento. Se as informações de configuração não forem localizadas, é fácil se esquecer de configurá-las ou, em alguns casos, nem mesmo estar ciente de que alguns recursos de proteção estão no sistema.

4. *Fornecer maneiras fáceis de corrigir vulnerabilidades de proteção.* Você deve incluir mecanismos simples para atualizar o sistema a fim de corrigir vulnerabilidades de proteção descobertas. Esses mecanismos podem incluir a verificação automática para as atualizações de proteção ou o download dessas atualizações assim que elas estiverem disponíveis. É importante que os usuários não possam ignorar esses mecanismos, pois eles, inevitavelmente, considerarão outros trabalhos como mais importantes. Existem vários exemplos de grandes problemas de proteção que surgiram (por exemplo, falha compelta de uma rede de hospital) porque os usuários não atualizaram seus softwares quando solicitado.

14.3 Sobrevivência de sistemas

Até este momento, discuti a engenharia de proteção do ponto de vista de uma aplicação em desenvolvimento. O adquirente e o desenvolvedor do sistema têm controle sobre todos os aspectos do sistema que podem ser atacados. Na realidade, tal como sugeri na Figura 14.1, os sistemas distribuídos modernos inevitavelmente contam com uma infraestrutura que inclui sistemas de prateleira e componentes reusáveis desenvolvidos por diferentes organizações. A proteção desses sistemas não depende apenas das decisões de projeto locais, mas também é afetada pela proteção de aplicações externas, pelos *web services* e pela infraestrutura da rede.

Ou seja, independentemente de quanta atenção seja dada à proteção, não podemos garantir que um sistema será capaz de resistir a ataques externos. Por conseguinte, para sistemas complexos de rede você deve assumir que a invasão é possível e que não é possível garantir a integridade do sistema. Então, você deve pensar em como fazer o sistema resiliente de modo que sobreviva para fornecer serviços essenciais aos usuários.

A capacidade de sobrevivência ou resistência (WESTMARK, 2004) é uma propriedade emergente de um sistema como um todo, e não uma propriedade de componentes individuais, que podem não ser capazes de sobreviver. A capacidade de sobrevivência de um sistema reflete sua capacidade de continuar a fornecer serviços de negócios essenciais ou de missão crítica para os usuários legítimos, enquanto o sistema está sob ataque ou após parte do sistema ter sido danificada. O dano pode ser causado por um ataque ou uma falha de sistema.

Os trabalhos sobre a capacidade de sobrevivência de sistema foram impulsionados pelo fato de que nossa vida econômica e social depende de uma infraestrutura crítica, controlada por computadores, a qual inclui a infraestrutura para a entrega de serviços públicos (energia, água, gás etc.), além da também crítica infraestrutura de entrega e gerenciamento de informações (telefones, Internet, serviços postais etc.). No entanto, a capacidade de sobrevivência não é uma simples questão de infraestruturas críticas. Qualquer organização baseada em sistemas críticos de computadores em rede deve estar preocupada com como seu negócio seria afetado se seus sistemas não sobrevivessem a um ataque mal-intencionado ou a uma falha catastrófica de sistema. Portanto, para sistemas críticos de negócios, o projeto e a análise de sobrevivência devem ser parte do processo de engenharia de proteção.

Manter a disponibilidade de serviços críticos é a essência da capacidade de sobrevivência. Isso significa que você precisa conhecer:

- os serviços de sistema mais críticos para o negócio;
- a qualidade mínima do serviço a ser mantida;
- como esses serviços podem ser comprometidos;
- como esses serviços podem ser protegidos;
- como se recuperar rapidamente caso os serviços se tornem indisponíveis.

Por exemplo, em um sistema que controla o envio de ambulâncias para chamadas de emergência, os serviços críticos são aqueles interessados em atender às chamadas e despachar as ambulâncias para a emergência médica. Outros serviços, como registro de chamadas e gerenciamento de localização de ambulâncias, são menos críticos, pois não exigem processamento em tempo real ou porque mecanismos alternativos podem ser usados. Por exemplo, para encontrar a localização de uma ambulância, você pode telefonar para algum membro da equipe e perguntar sua localização.

Ellison et al. (1999a; 1999b; 2002) desenvolveram um método de análise chamado Análise de Sobrevivência de Sistemas (*Survivable Systems Analysis*). Ele é usado para avaliar as vulnerabilidades em sistemas e apoiar o projeto de arquiteturas de sistema e recursos que promovam a capacidade de sobrevivência do sistema. Esses autores argumentam que obter a capacidade de sobrevivência depende de três estratégias complementares:

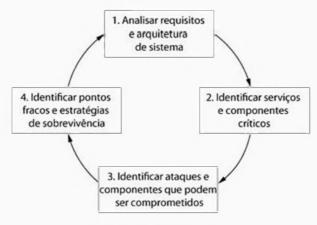
1. *Resistência*. Evitar problemas por meio da construção, no sistema, de habilidades que possam repelir ataques. Por exemplo, um sistema pode usar certificados digitais para autenticar usuários, o que dificulta o acesso de usuários não autorizados.
2. *Reconhecimento*. Detectar problemas por meio da construção, no sistema, de habilidades que possam repelir ataques e falhas e avaliar os danos resultantes. Por exemplo, os *checksums* podem ser associados a dados essenciais para que a corrupção desses dados possa ser detectada.
3. *Recuperação*. Tolerar problemas por meio da construção, no sistema, de habilidades para fornecer serviços essenciais quando sob ataque e recuperar a funcionalidade total após um ataque. Por exemplo, os mecanismos de tolerância a defeitos com diversas implementações da mesma funcionalidade podem ser incluídos para lidar com a perda de serviço de uma parte do sistema.

A análise de sobrevivência de sistemas é um processo de quatro estágios (Figura 14.7) que analisa os requisitos e a arquitetura do sistema atual e do proposto; identifica serviços críticos, cenários de ataque e 'pontos fracos' de sistema; também propõe alterações para melhoria da capacidade de sobrevivência de um sistema. As principais atividades, em cada um desses estágios, são as seguintes:

1. *Compreensão de sistema*. Para um sistema existente ou proposto, revisar suas metas (às vezes, chamadas objetivos de missão), seus requisitos e sua arquitetura.
2. *Identificação de serviços críticos*. Os serviços que sempre devem ser mantidos e os componentes necessários para manter esses serviços são identificados.
3. *Simulação de ataques*. Cenários ou casos de uso de possíveis ataques são identificados, junto com os componentes de sistema que serão afetados por esses ataques.
4. *Análise de sobrevivência*. Componentes essenciais e que podem ser comprometidos por um ataque são identificados, e estratégias de sobrevivência com base na resistência, no reconhecimento e na recuperação são identificadas.

Ellison et al. (1999b) apresentam um excelente estudo de caso do método, baseado em um sistema para oferecer suporte ao tratamento de saúde mental. Esse sistema é semelhante ao MHC-PMS que, neste livro, tenho usado como exemplo. Em vez de repetir sua análise, conforme mostrado na Figura 14.5, eu uso o sistema de negociação de ações para ilustrar alguns dos recursos da análise de sobrevivência.

Como você pode ver na Figura 14.5, esse sistema já tem alguma provisão para sobrevivência. As contas de usuários e os preços de ações são replicados entre os servidores para que as ordens possam ser dadas, mesmo que o servidor local esteja indisponível. Vamos supor que a capacidade dos usuários autorizados de colocar as ordens de ações seja o principal serviço a ser mantido. Para garantir que os usuários confiem no sistema, é essencial manter sua integridade. As ordens devem ser exatas e refletir as compras e vendas reais, feitas por um usuário do sistema.

Figura 14.7 Estágios na análise de sobrevivência

Para manter esse serviço de ordem, existem três componentes de sistema comumente usados:

1. *Autenticação de usuários*. Permite que os usuários autorizados façam *logon* no sistema.
2. *Cotação de preços*. Permite a cotação de preço de venda e compra de uma ação.
3. *Colocação de ordens*. Permite que ordens de compra e venda a determinado preço sejam feitas.

Esses componentes certamente fazem uso de ativos essenciais de dados, como um banco de dados de contas de usuário, um banco de dados de preços e um banco de dados de transações de ordens. Eles devem sobreviver a ataques, caso o serviço precise ser mantido.

Existem vários tipos diferentes de ataque ao sistema. Neste texto, podemos considerar duas possibilidades:

1. Um usuário mal-intencionado sente raiva de um usuário autorizado de um sistema. Ele ganha acesso ao sistema usando suas credenciais. São feitas ordens mal-intencionadas, e ações são compradas e vendidas, com a intenção de causar problemas para o usuário autorizado.
2. Um usuário não autorizado corrompe o banco de dados de transações, obtendo permissão para emitir diretamente comandos SQL. Portanto, é impossível a reconciliação de compras e vendas.

A Tabela 14.1 mostra exemplos de estratégias de resistência, reconhecimento e recuperação que podem ser usadas para ajudar a combater esses ataques.

Naturalmente, aumentar a capacidade de sobrevivência ou a resistência de um sistema custa dinheiro. Caso nunca tenham sofrido algum ataque que resultou em perdas, as empresas podem ser relutantes em investir em capacidade de sobrevivência. No entanto, é melhor comprar boas fechaduras e um alarme antes de sua casa ser assaltada, e não depois; é melhor investir na capacidade de sobrevivência, antes e não após um ataque bem-sucedido. A análise de sobrevivência ainda não é parte da maioria dos processos de engenharia de software, mas, como cada vez mais sistemas se tornam sistemas críticos de negócios, as análises podem ficar mais amplamente usadas.

Tabela 14.1 Análise de sobrevivência em um sistema de negociação de ações

Ataque	Resistência	Reconhecimento	Recuperação
Usuário não autorizado coloca ordens maliciosas	Exigir uma senha de tratamento diferente da senha de *login* para colocar ordens.	Enviar cópia da ordem, por e-mail, para o usuário autorizado, com telefone de contato (para que se possam detectar ordens maliciosas). Manter o histórico da ordem do usuário e verificar se existem padrões anormais de negociação.	Oferecer mecanismos para 'desfazer' automaticamente negociações e restaurar contas de usuários. Reembolsar os usuários pelos prejuízos devidos à negociação mal-intencionada. Proteger-se contra perdas derivadas.
Corrupção de banco de dados de transações	Exigir que usuários privilegiados sejam autorizados a usar um mecanismo de autenticação mais forte, como os certificados digitais.	Manter cópias — somente leitura — das operações de um escritório em um servidor internacional. Periodicamente, comparar as transações para verificar se há corrupção. Manter a soma criptográfica de todos os registros de transações para se detectar possível corrupção.	Recuperar banco de dados a partir de cópias de *backup*. Fornecer um mecanismo para reproduzir transações de um momento específico para recriar o banco de dados de transações.

PONTOS IMPORTANTES

- A engenharia de proteção concentra-se em desenvolver e manter sistemas de software capazes de resistir a ataques mal-intencionados destinados a danificar um sistema baseado em computadores, bem como seus dados.
- As ameaças à proteção podem ameaçar a confidencialidade, a integridade ou a disponibilidade de um sistema ou de seus dados.
- O gerenciamento de riscos de proteção envolve avaliar os prejuízos resultantes de ataques a um sistema e derivar os requisitos de proteção destinados a eliminar ou reduzir perdas.
- O projeto para proteção envolve uma arquitetura de sistema protegida, seguindo boas práticas para o projeto de sistemas protegidos e incluindo funcionalidade que minimize a possibilidade de introdução de vulnerabilidades quando o sistema for implantado.
- As principais questões, ao se projetar uma arquitetura de sistemas protegida, incluem organizar a estrutura de sistema para proteger os principais ativos e distribuir os ativos de sistema para minimizar as perdas resultantes de um ataque bem-sucedido.
- As diretrizes de projeto de proteção sensibilizam os projetistas para questões de proteção que possam ter sido desconsideradas. Estas fornecem uma base para criação de *ckecklists* de revisão de proteção.
- No suporte à implantação de proteção, deve-se fornecer uma maneira de exibição e análise das configurações de sistema, localizar as definições de configuração para que as configurações importantes não sejam esquecidas, minimizar privilégios *default* atribuídos aos usuários de sistema e fornecer maneiras de correção das vulnerabilidades de proteção.
- A sobrevivência de sistema é a capacidade de um sistema de continuar a entregar serviços essenciais de negócios ou de missão crítica para usuários legítimos, enquanto ele está sob ataque ou após parte do sistema ter sido danificada.

LEITURA COMPLEMENTAR

'Survivable Network System Analysis: A Case Study.' Um excelente artigo que introduz a noção de sobrevivência do sistema; usa um estudo de caso de um sistema de registro de tratamento de saúde mental para ilustrar a aplicação de um método de sobrevivência. (ELLISON, R. J.; LINGER, R. C.; LONGSTAFF, T.; MEAD, N. R. *IEEE Software*, v. 16, n. 4, jul./ago. 1999.)

Building Secure Software: How to Avoid Security Problems the Right Way. Um bom livro prático que discute a proteção a partir de uma perspectiva de programação. (VIEGA, J.; McGRAW, G. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2002.)

Security Engineering: A Guide to Building Dependable Distributed Systems. Essa é uma discussão aprofundada e abrangente sobre os problemas de construção de sistemas protegidos. É centrada em sistemas, e não em engenharia de software, com ampla cobertura de hardware e rede, com excelentes exemplos retirados de falhas de sistemas reais. (ANDERSON, R. *Security Engineering*: A Guide to Building Dependable Distributed Systems. 2. ed. John Wiley & Sons, 2008.)

EXERCÍCIOS

14.1 Explique as diferenças importantes entre a engenharia de proteção de aplicação e a engenharia de proteção de infraestrutura.

14.2 Para o MHC-PMS, sugira um exemplo de ativo, exposição, vulnerabilidade, ataque, ameaça e controle.

14.3 Explique por que existe a necessidade de a avaliação de riscos ser um processo contínuo, desde os estágios de engenharia de requisitos até o uso operacional de um sistema.

14.4 Usando suas respostas para a questão 14.2, sobre o MHC-PMS, avalie os riscos associados a esse sistema e proponha dois requisitos de sistema que possam reduzir esses riscos.

14.5 Explique, usando uma analogia de um contexto de engenharia não ligada a software, por que uma abordagem em camadas deve ser usada para proteger os ativos.

14.6 Explique por que, em situações nas quais a disponibilidade de sistema é fundamental, é importante usar diversas tecnologias para oferecer suporte aos sistemas distribuídos.

14.7 O que é a engenharia social? Por que, em uma grande organização, é difícil proteger-se contra ela?

14.8 Para qualquer sistema de software de prateleira (por exemplo, Microsoft Word), analise os recursos de configuração incluídos e discuta todos os problemas que você encontrar.

14.9 Explique como as estratégias complementares de resistência, reconhecimento e recuperação podem ser usadas para aumentar a capacidade de sobrevivência de um sistema.

14.10 Para o sistema de negociação de ações discutido na Seção 14.2.1, cuja arquitetura é mostrada na Figura 14.5, sugira dois ataques plausíveis ao sistema, e proponha estratégias que poderiam conter esses ataques.

REFERÊNCIAS

ALBERTS, C.; DOROFEE, A. *Managing Information Security Risks*: The OCTAVE Approach. Boston: Addison-Wesley, 2002.

ALEXANDER, I. Misuse Cases: Use Cases with Hostile Intent. *IEEE Software*, v. 20, n. 1, 2003, p. 58-66.

ANDERSON, R. *Security Engineering*. 2. ed. Chichester: John Wiley & Sons, 2008.

BERGHEL, H. The Code Red Worm. *Comm. ACM*, v. 44, n. 12, 2001, p. 15-19.

BISHOP, M. *Introduction to Computer Security*. Boston: Addison-Wesley, 2005.

CRANOR, L.; GARFINKEL, S. *Security and Usability*: Designing secure systems that people can use. Sebastopol, Calif.: O'Reilly Media Inc., 2005.

ELLISON, R.; LINGER, R.; LIPSON, H.; MEAD, N.; MOORE, A. Foundations of Survivable Systems Engineering. *Crosstalk*: The Journal of Defense Software Engineering, v. 12, 2002, p. 10-15.

ELLISON, R. J.; FISHER, D. A.; LINGER, R. C.; LIPSON, H. F.; LONGSTAFF, T. A.; MEAD, N. R. Survivability: Protecting Your Critical Systems. *IEEE Internet Computing*, v. 3, n. 6, 1999a, p. 55-63.

ELLISON, R. J.; LINGER, R. C.; LONGSTAFF, T.; MEAD, N. R. Survivable Network System Analysis: A Case Study. *IEEE Software*, v. 16, n. 4, 1999b, p. 70-77.

PFLEEGER, C. P.; PFLEEGER, S. L. *Security in Computing*. 4. ed. Boston: Addison-Wesley, 2007.

SCHNEIER, B. *Secrets and Lies*: Digital Security in a Networked World. Nova York: John Wiley & Sons, 2000.

SINDRE, G.; OPDAHL, A. L. Eliciting Security Requirements through Misuse Cases. *Requirements Engineering*, v. 10, n. 1, 2005, p. 34-44.

SPAFFORD, E. The Internet Worm: Crisis and Aftermath. *Comm ACM*, v. 32, n. 6, 1989, p. 678-687.

VIEGA, J.; McGRAW, G. *Building Secure Software*. Boston: Addison-Wesley, 2002.

WESTMARK, V. R. A Definition for Information System Survivability. 37th Havai Int. Conf. on System Sciences, Havai, 2004, p. 903-1003.

WHEELER, D. A. *Secure Programming for Linux and UNix HOWTO*, 2003. Publicação digital, disponível em: <http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html>.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 **15** 16 17 18 19 20 21 22 23 24 25 26

Garantia de confiança e proteção

Objetivos

O objetivo deste capítulo é descrever as técnicas de verificação e validação no desenvolvimento de sistemas críticos. Com a leitura deste capítulo, você:

- compreenderá como diferentes abordagens da análise estática podem ser usadas na verificação de sistemas críticos de software;
- compreenderá os princípios dos testes de confiabilidade e proteção, e os problemas inerentes de se testar sistemas críticos;
- saberá por que a garantia do processo é importante, especialmente para o software que necessita ser certificado por um regulador;
- será apresentado a casos de segurança e confiança que apresentam argumentos e evidências de segurança e confiança de sistema.

Conteúdo

A garantia de confiança e proteção busca verificar que um sistema crítico atende a seus requisitos de confiança. Isso requer processos de verificação e validação (V & V) que procurem erros de especificação, de projeto e de programa que possam afetar a disponibilidade, a segurança, a confiabilidde ou a proteção de um sistema.

A verificação e validação de um sistema crítico têm muito em comum com a validação de qualquer outro sistema de software. Os processos de V & V devem demonstrar que o sistema atende a sua especificação e que os serviços e o comportamento do sistema suportam os requisitos do cliente. Dessa forma, geralmente, descobrem erros de requisitos e projeto e defeitos de programas que precisam ser reparados. Entretanto, os sistemas críticos requerem testes e análises mais rigorosos. São duas as razões para tanto:

1. *Custos de falhas*. Os custos e as consequências de falhas de sistemas críticos são potencialmente muito mais altos do que os de sistemas não críticos. Para reduzir esses custos, é necessário investir na verificação e validação de sistema. Geralmente, é mais econômico encontrar e remover defeitos antes de o sistema ser entregue do que pagar por custos resultantes dos acidentes ou interrupções dos serviços de sistema.
2. *Validação de atributos de confiança*. É possível que você precise criar um exemplo formal para os clientes e reguladores, a fim de demonstrar que o sistema atende a seus requisitos de confiança (disponibilidade, confiabilidade, segurança e proteção). Em alguns casos, os reguladores externos, como autoridades nacionais de aviação, podem precisar certificar a segurança do sistema antes que ele seja implantado. Para obter essa certificação, é necessário

demonstrar como o sistema foi validado. Para tanto, você pode ter de projetar e efetuar os procedimentos especiais de V & V que coletem evidências sobre a confiança do sistema.

Por essas razões, geralmente os custos de verificação e validação para sistemas críticos são muito mais elevados do que para outras classes de sistemas. Normalmente, mais da metade dos custos de desenvolvimento de um sistema crítico são gastos em V & V.

Embora os custos de V & V sejam elevados, são normalmente justificados, pois são significativamente menores do que as perdas de um acidente. Por exemplo, em 1996 aconteceu a falha um sistema de software de missão crítica no foguete Ariane 5 e diversos satélites foram destruídos. Ninguém foi ferido, mas as perdas totais desse acidente foram centenas de milhões de dólares. O inquérito subsequente descobriu que as deficiências no V & V de sistema foram, em parte, responsáveis pela falha. Algumas revisões mais eficazes, relativamente baratas, poderiam ter descoberto o problema que causou o acidente.

Embora o foco preliminar da garantia de confiança e proteção esteja na validação do sistema em si, as atividades relacionadas devem verificar se o processo definido para o desenvolvimento de sistema está sendo seguido. Como expliquei no Capítulo 13, a qualidade do sistema é afetada pela qualidade dos processos usados no desenvolvimento do sistema. Para resumir, bons processos geram bons sistemas.

Os resultados do processo de garantia de confiança e proteção são evidências tangíveis, assim como relatórios de revisão, resultados de teste etc., sobre a confiança de um sistema. Essas evidências podem ser usadas para justificar a decisão de que esse sistema é confiável o bastante para ser implantado e usado. Às vezes, a evidência de confiança de sistema é montada em um exemplo de confiança ou de segurança. Esta pode ser usada para convencer um cliente ou regulador externo de que a confiança do desenvolvedor na confiança ou segurança de um sistema é justificada.

15.1 Análise estática

As técnicas de análise estática são técnicas de verificação de sistema que não envolvem a execução de um programa. Elas trabalham em uma representação da fonte do software ou outro modelo de especificação ou de projeto, ou no código-fonte do programa. As técnicas de análise estática podem ser usadas para verificar os modelos de especificação — e de projeto de um sistema — para encontrar erros antes que uma versão executável do sistema esteja disponível. Ainda tem a vantagem de que a presença de erros não interrompe a verificação do sistema. Quando você testa um programa, os defeitos podem mascarar ou esconder outros defeitos e, assim, é necessário remover os defeitos detectados e, então, repetir o processo de testes.

Como discutido no Capítulo 8, talvez a técnica de análise estática mais usada seja a revisão e inspeção em pares, em que as especificações, o projeto ou um programa são verificados por um grupo de pessoas que examinam, em detalhes, o projeto ou o código, procurando possíveis erros ou omissões. Outra técnica é usar ferramentas de modelagem de projeto para verificar se existem anomalias na UML, como um mesmo nome sendo usado para objetos diferentes. Entretanto, para sistemas críticos, técnicas adicionais de análise estática podem ser usadas:

1. *Verificação formal*, em que você produz, matematicamente, argumentos rigorosos para que um programa atenda a suas especificações.
2. *Verificação de modelos*, em que um provador de teoremas é usado para verificar uma descrição formal do sistema, para ver se existem inconsistências.
3. *Análise automatizada de programa*, em que o código-fonte de um programa é verificado contra padrões conhecidos de erros potenciais.

Essas técnicas estão estreitamente relacionadas. A verificação de modelos conta com um modelo formal de sistema que pode ser criado por meio de uma especificação formal. Os analisadores estáticos podem usar afirmações formais embutidas em um programa como comentários para certificar-se de que o código associado seja consistente com essas afirmações.

15.1.1 Verificação e métodos formais

Os métodos formais de desenvolvimento de software, conforme discutido no Capítulo 12, contam com um modelo formal de sistema que serve como uma especificação. Esses métodos formais são relacionados, principal-

mente, a uma análise matemática da especificação, bem como à transformação da especificação em uma representação mais detalhada, semanticamente equivalente, ou, ainda, à verificação formal, em que uma representação do sistema é semanticamente equivalente a outra representação.

No processo de V & V, os métodos formais podem ser usados em diferentes estágios:

1. Uma especificação formal do sistema pode ser desenvolvida e matematicamente analisada por inconsistências. Essa técnica é eficaz para descobrir erros e omissões de especificação. A verificação de modelos, discutida na seção seguinte, é uma abordagem para análise de especificação.
2. Você pode verificar formalmente, usando argumentos matemáticos, se o código de um sistema de software é consistente com sua especificação. Isso requer uma especificação formal. É eficaz em descobrir erros de programação e alguns erros de projeto.

Em virtude do *gap* semântico entre uma especificação formal de sistema e um código de programa, é difícil provar que um programa desenvolvido separadamente é consistente com sua especificação. Consequentemente, o trabalho de verificação de um programa é, atualmente, baseado no desenvolvimento transformacional. Em um processo de desenvolvimento transformacional, uma especificação formal é transformada em uma série de representações até um código de programa. As ferramentas de software suportam o desenvolvimento das transformações e ajudam a verificar se as representações correspondentes do sistema são consistentes. O método B é, provavelmente, o método transformacional formal mais extensamente usado (ABRIAL, 2005; WORDSWORTH, 1996); ele foi usado para o desenvolvimento de sistemas de controle de trens e softwares aviônicos.

Os proponentes dos métodos formais afirmam que o uso desses métodos garante sistemas mais seguros e confiáveis. A verificação formal demonstra que o programa desenvolvido cumpre suas especificações e que erros de implementação não comprometerão a confiança do sistema. Se você desenvolver um modelo formal de sistemas concorrentes usando uma especificação escrita em uma linguagem, tal como a CSP (SCHNEIDER, 1999), poderá descobrir condições que podem resultar em um *deadlock* no programa final, além de ser capaz de resolvê-los. Isso é muito difícil quando você está testando sozinho.

Entretanto, uma especificação e prova formais não garantem que o software será confiável no uso prático. As razões para isso são:

1. A especificação não pode refletir os requisitos reais dos usuários de sistema. Como discutido no Capítulo 12, os usuários de sistema raramente compreendem as notações formais e, assim, não podem ler diretamente a especificação formal para encontrar erros e omissões. Isso significa que existe uma alta probabilidade de a especificação formal conter erros e de ela não ser uma representação exata dos requisitos de sistema.
2. A prova pode conter erros. As provas de programas são grandes e complexas e, assim como os programas grandes e complexos, geralmente contêm erros.
3. A prova pode fazer suposições incorretas sobre a maneira como o sistema será usado. Se o sistema não for usado como previsto, a prova pode ser inválida.

A verificação de um sistema de software não trivial toma muito tempo, requer perícia matemática e ferramentas especializadas de software, como provadores de teoremas. Consequentemente, esse é um processo caro e, como o tamanho do sistema aumenta, os custos de verificação formal aumentam desproporcionalmente. Consequentemente, muitos engenheiros de software pensam que a verificação formal não é eficaz. Eles acreditam que o mesmo nível de confiança no sistema pode ser alcançado de forma mais barata, com outras técnicas de validação, como inspeções e testes de sistema.

Independentemente de suas desvantagens, minha percepção é de que os métodos formais e a verificação formal têm um papel importante no desenvolvimento de sistemas críticos de software. As especificações formais são muito eficazes em descobrir aqueles problemas de especificação que são as causas mais comuns de falhas de sistema. Embora a verificação formal para sistemas grandes seja ainda pouco prática, pode ser usada para verificar componentes críticos de segurança e de proteção.

15.1.2 Verificação de modelo

A verificação formal de programas por meio de uma abordagem dedutiva é difícil e cara, mas as abordagens alternativas à análise formal foram desenvolvidas — e são baseadas em uma noção mais restrita de correção. Entre elas, a abordagem mais bem-sucedida é chamada verificação de modelos (BAIER e KATOEN, 2008). Essa abordagem foi usada na verificação de projetos de sistemas de hardware e tem sido extensivamente usada em sistemas

críticos de software como o software de controle de veículos de exploração de Marte da NASA (REGAN e HAMILTON, 2004) e o software processador de chamadas telefônicas (CHANDRA et al., 2002).

A verificação de modelos envolve a criação de um modelo de um sistema e a verificação da correção desse modelo com o uso de ferramentas especializadas de software. Muitas ferramentas diferentes de verificação de modelos foram desenvolvidas. Para softwares, provavelmente, a mais usada é a SPIN (HOLZMANN, 2003). Os estágios envolvidos na verificação de modelos são mostrados na Figura 15.1.

O processo de verificação de modelos envolve a construção de um modelo formal de um sistema, geralmente como uma máquina de estado finito estendida. Os modelos são expressos em qualquer que seja o sistema de verificação de modelo utilizado — por exemplo, o verificador de modelo SPIN usa uma linguagem chamada Promela. Um conjunto de propriedades desejáveis de sistema é identificado e escrito em uma notação formal, geralmente baseada na lógica temporal. Um exemplo de tal propriedade no sistema metereológico no deserto pode ser que o sistema sempre alcance o estado 'transmitindo' a partir do estado 'gravando'.

O verificador de modelos explora todos os caminhos pelo modelo (isto é, todas as possíveis transições de estado), verificando que a propriedade se mantém em cada caminho. Caso isso aconteça, o verificador de modelos confirma que o modelo está correto com respeito a essa propriedade. Caso não se mantenha para um caminho particular, o verificador de modelos gera um contra-exemplo que ilustra onde a propriedade não é verdadeira. A verificação de modelos é particularmente útil na validação de sistemas concorrentes, notoriamente difíceis de se testar em virtude de sua sensibilidade a tempo. O verificador pode explorar transições intercaladas e concorrentes e descobrir problemas potenciais.

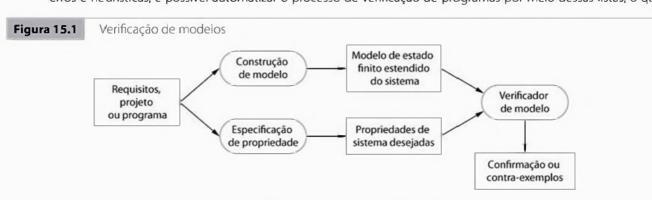
Uma questão central na verificação de modelos é a criação do modelo de sistemas. Se o modelo precisa ser criado manualmente (do documento de requisitos ou de projeto), será um processo caro, pois a criação do modelo será demorada. Além disso, existe a possibilidade de que o modelo criado não seja um modelo preciso dos requisitos ou projeto. Consequentemente, é melhor se o modelo puder ser criado automaticamente, a partir do código-fonte do programa. O sistema Java Pathfinder (VISSER et al., 2003) é um exemplo de um sistema verificador de modelos que trabalha diretamente a partir de uma representação de código Java.

Computacionalmente, a verificação de modelos é muito cara, pois usa uma abordagem exaustiva para verificar todos os caminhos do modelo de sistema. Enquanto o tamanho do sistema aumenta, também aumenta o número de estados, e, consequentemente, o número de caminhos a ser verificado. Isso significa que, para sistemas grandes, a verificação de modelos pode ser pouco prática, devido ao tempo de computador requerido para executar as verificações.

Entretanto, assim como melhoram os algoritmos para identificar essas partes do estado que não foram exploradas, para verificar uma propriedade particular fica cada vez mais prático usar verificação de modelos, rotineiramente, no desenvolvimento de sistemas críticos. Isso realmente não é aplicável aos sistemas organizacionais orientados a dados, mas pode ser usado para se verificar os sistemas de software embutidos, modelados como máquinas de estado.

15.1.3 Análise estática automática

Como discutido no Capítulo 8, as inspeções de programa são, frequentemente, dirigidas por *checklists* de verificação de erros e heurísticas. Elas identificam erros comuns em diferentes linguagens de programação. Para alguns erros e heurísticas, é possível automatizar o processo de verificação de programas por meio dessas listas, o que

Figura 15.1 Verificação de modelos

tem resultado no desenvolvimento dos analisadores estáticos automatizados capazes de encontrar fragmentos de código que possam estar incorretos.

As ferramentas de análise estática trabalham no código-fonte de um sistema e, pelo menos para alguns tipos de análise, nenhuma entrada adicional é requerida. Isso significa que os programadores não precisam aprender notações especializadas para escrever as especificações de programa, de maneira que os benefícios da análise podem ser imediatamente esclarecidos. Isso torna a análise estática automatizada mais fácil de ser introduzida em um processo de desenvolvimento do que a verificação formal ou verificação de modelos. Provavelmente, essa é a técnica de análise estática mais extensamente usada.

Os analisadores estáticos automatizados são as ferramentas de software que fazem a varredura do texto-fonte de um programa e detectam possíveis defeitos e anomalias. Eles analisam o texto do programa e, assim, reconhecem os diferentes tipos de declarações em um programa. Podem detectar se as declarações estão bem formadas, fazem inferências sobre o fluxo de controle no programa e, em muitos casos, calculam o conjunto de todos os possíveis valores de dados do programa. Eles complementam os recursos de detecção de erros fornecidos pelo compilador de linguagem e podem ser usados como parte do processo de inspeção ou como uma atividade separada do processo de V & V. A análise estática automatizada é mais rápida e mais barata do que as revisões de código detalhadas. Entretanto, estas não podem descobrir algumas classes de erros que poderiam ser identificados em reuniões de inspeção de programa.

A intenção da análise estática automática é desenhar um código de atenção do leitor para as anomalias no programa, como as variáveis usadas sem iniciação, sem uso, ou ainda, nos dados, cujos valores poderiam sair dos limites. Os exemplos de problemas que podem ser detectados pela análise estática são mostrados na Tabela 15.1. Naturalmente, as verificações específicas são de linguagem de programação específica e dependem do que é e do que não é permitido na linguagem. Frequentemente, as anomalias resultam de omissões ou erros de programação, assim que elas destacam o que poderia dar errado quando o programa é executado. Entretanto, você deve compreender que essas anomalias não são necessariamente defeitos de programa; podem ser construções deliberadas, introduzidas pelo programador, ou ainda, a anomalia pode não ter nenhuma consequência adversa.

Em analisadores estáticos, há três níveis de verificação que podem ser implementados:

1. *Verificação de erros característicos*. Nesse nível, o analisador estático conhece erros comuns, realizados por programadores em linguagens como Java ou C. A ferramenta analisa o código, buscando por padrões que são característicos desses problemas e destaca-os para o programador. Embora relativamente simples, a análise baseada em erros comuns pode ser muito efetiva. Zheng et al. (2006) estudaram o uso da análise estática contra uma grande base de código em C e em C++ e descobriram que 90% dos erros dos programas resultaram de dez tipos de erros característicos.

Tabela 15.1 Verificação automatizada de análise estática

Classe de defeito	Verificação de análise estática
Defeitos de dados	Variáveis usadas antes da iniciação Variáveis declaradas, mas nunca usadas Variáveis atribuídas duas vezes, mas nunca usadas entre atribuições Possíveis violações de limites de vetor Variáveis não declaradas
Defeitos de controle	Código inacessível Ramos incondicionais dentro de *loops*
Defeitos de entrada/saída	Saída de variáveis duas vezes sem atribuição intermediária
Defeitos de interface	Incompatibilidades de tipo de parâmetro Incompatibilidades de número de parâmetros Não uso de resultados de funções Funções e procedimentos não chamados
Defeitos de gerenciamento de armazenamento	Ponteiros não atribuídos Ponteiro aritmético Perdas de memória

2. *Verificação de erros definidos pelo usuário.* Nessa abordagem, os usuários do analisador estático podem definir padrões de erros, estendendo os tipos de erro que podem ser detectados, o que é particularmente útil nas situações em que os requisitos devem ser mantidos (por exemplo, o método A deve ser chamado sempre antes do método B). Assim, uma organização pode coletar informações sobre os erros comuns que ocorrem em seus programas e estenderem as ferramentas de análise estática para destacar esses erros.
3. *Verificação de asserções.* Essa é a abordagem mais geral e mais poderosa da análise estática. Os desenvolvedores incluem asserções formais (frequentemente, escritas como comentários estilizados) em seus programas que definem os relacionamentos que devem manter naquele ponto de programa. Por exemplo, pode ser incluída uma asserção que indique que o valor de uma variável deve se encontrar no intervalo de x..y. O analisador executa simbolicamente o código e destaca as declarações em que a asserção não pode ser mantida. Essa abordagem é usada por analisadores como Splint (EVANS e LA ROCHELLE, 2002) e o SPARK Examiner (CROXFORD e SUTTON, 2006).

A análise estática é eficaz para encontrar erros em programas, mas, frequentemente, gera um grande número de 'falsos positivos'. Estes são seções de código nas quais não existem erros, mas em que as regras do analisador estático detectaram um potencial para erros. O número de falsos positivos pode ser reduzido adicionando-se mais informação ao programa na forma de asserções, mas, obviamente, esse processo requer trabalho adicional do desenvolvedor do código. Deve ser feito um trabalho para exibir esses falsos positivos antes que o próprio código possa ser verificado para ver se existem erros.

A análise estática é particularmente valiosa para verificações de proteção (EVANS e LA ROCHELLE, 2002). Os analisadores estáticos podem ser customizados para verificar problemas bem conhecidos, tais como o *overflow* de *buffer* ou entradas não verificadas, que podem ser exploradas por invasores. A verificação por problemas bem conhecidos é eficaz em melhorar a proteção, pois a maioria dos invasores baseia seus ataques nas vulnerabilidades comuns.

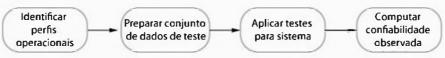
Como discuto adiante, os testes de proteção são difíceis, pois os invasores costumam tomar atitudes inesperadas, difíceis de serem antecipadas pelos testadores. Os analisadores estáticos podem incorporar capacidade detalhada de proteção, que os testadores podem não ter e que pode ser aplicada antes que um programa seja testado. Se você usar a análise estática, poderá fazer afirmações que são verdadeiras para todas as execuções de programa possíveis, não apenas aquelas que correspondem aos testes que você projetou.

Atualmente, a análise estática é usada rotineiramente por muitas organizações em seus processos de desenvolvimento de software. A Microsoft introduziu a análise estática no desenvolvimento de drivers de dispositivos (LARUS et al., 2003), nos quais as falhas de programa podem ter sérios efeitos. Eles têm estendido a abordagem por meio de uma gama mais ampla de seus softwares, para procurar problemas de proteção, assim como por erros que afetem a confiabilidade de programa (BALL et al., 2006). Como parte do processo de V & V (NGUYEN e OURGHANLIAN, 2003), rotineiramente, muitos sistemas críticos, incluindo os aviônicos e os sistemas nucleares, são analisados estaticamente.

15.2 Testes de confiabilidade

O teste de confiabilidade é um processo de teste cujo objetivo é medir a confiabilidade de um sistema. Como expliquei no Capítulo 10, existem diversas métricas de confiabilidade, como a POFOD, probabilidade de falha sob demanda, e a ROCOF, taxa de ocorrência de falha. Elas podem ser usadas para especificar a confiabilidade requerida de software, quantativamente. Caso o sistema alcance o nível requerido de confiabilidade, você pode verificar o processo de teste de confiabilidade.

O processo de medição de confiabilidade de um sistema é ilustrado na Figura 15.2. Esse processo envolve quatro estágios:

Figura 15.2 Medição de confiabilidade

1. Você começa estudando os sistemas do mesmo tipo já existentes, para compreender como eles são usados na prática. Isso é importante porque você está tentando medir a confiabilidade como ela é vista por um usuário de sistema. Seu objetivo é definir um perfil operacional, e um perfil operacional identifica classes de entradas de sistema e a probabilidade de que essas entradas ocorram com uso normal.
2. Então você constrói um conjunto de dados de teste que reflita o perfil operacional. Isso significa que você cria dados de teste com a mesma distribuição de probabilidade que os dados de teste para os sistemas que você estudou. Geralmente, você usará um gerador de dados de teste para suportar esse processo.
3. Você testa o sistema usando esses dados e conta o número e o tipo de falhas que ocorrem. Os tempos dessas falhas também são registrados. Como discutido no Capítulo 10, as unidades de tempo escolhidas devem ser apropriadas para a métrica de confiabilidade usada.
4. Depois que você observou um número de falhas estatisticamente significativo, você pode computar a confiabilidade de software e encontrar o valor apropriado de métrica de confiabilidade.

Essa abordagem de quatro passos, às vezes, é chamada 'teste estatístico'. O objetivo do teste estatístico é avaliar a confiabilidade de sistema. Isso contrasta com os testes de defeitos, discutidos no Capítulo 8, para os quais o objetivo é descobrir defeitos de sistema. Prowell et al. (1999) dão uma boa descrição de testes estatísticos em seu livro sobre a engenharia de software Cleanroom.

Essa abordagem conceitualmente atrativa para a medição de confiabilidade, na prática, não é de fácil aplicação. As principais dificuldades que surgem são:

1. *Incerteza de perfil operacional.* Os perfis operacionais baseados na experiência com outros sistemas não podem ser uma reflexão exata do uso real do sistema.
2. *Custos elevados de geração de dados de teste.* Pode ser muito caro gerar o grande volume de dados requeridos por um perfil operacional, a menos que o processo possa ser totalmente automatizado.
3. *Incertezas estatísticas quando alta confiabilidade é especificada.* Você precisa gerar um número de falhas estatisticamente significativo que permita medir a confiabilidade acuradamente. Quando o software já é confiável, ocorrem poucas falhas e é difícil gerar falhas novas.
4. *Reconhecimento de falhas.* Nem sempre é óbvio se uma falha de sistema ocorreu ou não. Se você tiver uma especificação formal, poderá identificar desvios dessa especificação, mas, se ela estiver em linguagem natural, pode haver ambiguidades que signifiquem que os observadores venham a discordar se o sistema falhou.

De longe, a melhor maneira de gerar o grande conjunto de dados requerido para medir a confiabilidade é usar um gerador de dados de teste, que possa ser ajustado até gerar automaticamente as entradas que combinem com o perfil operacional. Entretanto, em geral não é possível automatizar a produção de todos os dados de teste para sistemas interativos, pois, frequentemente, as entradas são uma resposta às saídas de sistema. Conjuntos de dados para esses sistemas precisam ser gerados manualmente, com custos mais elevados. Mesmo onde a automatização completa seja possível, a escrita de comandos para o gerador de dados de teste pode tomar uma significativa quantidade de tempo.

Os testes estatísticos podem ser usados em conjunto com uma injeção de defeitos, para obtenção de dados sobre quão eficaz tem sido o processo de testes de defeitos. A injeção de defeitos (VOAS, 1997) é a injeção deliberada de erros em um programa. Quando o programa é executado, eles conduzem a defeitos de programa e a falhas associadas. Então, você analisa a falha para descobrir se a causa-raiz foi um dos erros que você adicionou ao programa. Se você encontrar que X% dos defeitos injetados conduzem a falhas, os proponentes da injeção de defeitos argumentam que o processo de teste de defeitos também terá descoberto X% de defeitos reais no programa.

Naturalmente, isso supõe que a distribuição e o tipo de defeitos injetados, na prática, são compatíveis com os defeitos que surgem. É razoável pensar que isso pode ser verdadeiro para defeitos advindos de erros de programação, mas a injeção de defeitos não é eficaz em prever o número de defeitos resultantes de erros de requisitos ou de projeto.

Frequentemente, os testes estatísticos revelam erros no software que não foram descobertos por outros processos de V & V. Esses erros podem significar que a confiabilidade de um sistema está aquém dos requisitos e que reparos devem ser feitos. Depois de finalizar os reparos, o sistema pode ser testado novamente para reavaliar sua confiabilidade. Depois que esse processo de reparo e de novo teste seja repetido diversas vezes, pode ser possível extrapolar os resultados e predizer quando algum nível requerido de confiabilidade será alcançado, o que requer o ajuste de dados extrapolados para um modelo de crescimento de confiabilidade, que mostra como a confiabili-

dade tende a melhorar ao longo do tempo. Isso ajuda no planejamento de testes. Às vezes, um modelo de crescimento pode revelar que o nível requerido de confiabilidade nunca será alcançado, motivo pelo qual os requisitos devem ser renegociados.

15.2.1 Perfis operacionais

O perfil operacional de um sistema de software reflete como ele será usado na prática. Consiste em uma especificação das classes de entradas e da probabilidade de sua ocorrência. Quando um novo sistema de software substitui um sistema automatizado existente, é razoavelmente fácil avaliar o provável padrão de uso do novo software. Este deve corresponder ao uso existente, com alguma permissão para a funcionalidade nova que (presumidamente) está incluída no software novo. Por exemplo, um perfil operacional pode ser especificado para sistemas de comutação telefônica, pois as companhias de telecomunicação sabem dos testes-padrão de chamada que esses sistemas precisam tratar.

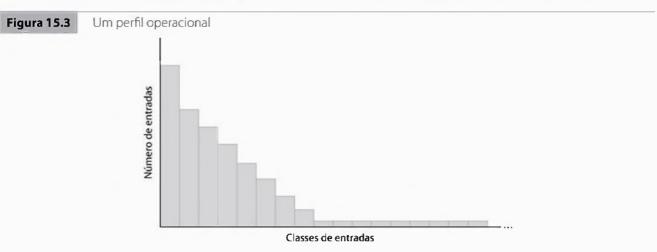
Tipicamente, o perfil operacional é tal que as entradas cuja probabilidade de serem geradas são mais elevada caem em um pequeno número de classes, como mostrado no lado esquerdo da Figura 15.3. Existe um grande número de classes em que as entradas são altamente improváveis, mas não impossíveis. Estas são mostradas à direita na Figura 15.3. A elipse (...) significa que existem muito mais dessas entradas incomuns do que as que são mostradas.

Musa (1998) discute o desenvolvimento de perfis operacionais em sistemas de telecomunicação. Como existe uma longa história de coleta de dados de uso para esse domínio, o processo de desenvolvimento de perfil operacional é relativamente direto. Ele simplesmente reflete os dados históricos de uso. Para um sistema que exigia cerca de 15 pessoas-ano de esforço de desenvolvimento, um perfil operacional foi desenvolvido em cerca de uma pessoa-mês. Em outros casos, a geração de perfil operacional levou mais tempo (duas a três pessoas-ano), mas o custo foi distribuído em vários releases de sistema. Musa conta que sua companhia teve retorno de pelo menos 10-x o retorno sobre o investimento requerido para desenvolver um perfil operacional.

Entretanto, quando um sistema de software é novo e inovativo, é difícil antecipar como ele será usado. Consequentemente, é praticamente impossível criar um perfil operacional exato. Diversos usuários com expectativas, conhecimentos e experiências diferentes podem usar o sistema novo. Não existe uma base histórica de dados de uso. Esses usuários podem empregar o sistema de maneiras não antecipadas pelos desenvolvedores de sistema.

Desenvolver um perfil operacional exato é possível para alguns tipos de sistema, como os sistemas de telecomunicação, que têm um padrão de uso. Entretanto, para outros tipos de sistema, existem muitos usuários diferentes, cada um com suas próprias maneiras de usar o sistema. Como discutido no Capítulo 10, diferentes usuários podem ter impressões completamente diferentes da confiabilidade, pois usam o sistema de maneiras diferentes.

O problema é agravado porque os perfis operacionais não são estáticos, mas mudam à medida que o sistema é usado. Na medida em que os usuários aprendem sobre o novo sistema, eles confiam mais nele e começam a usá-lo de maneiras mais sofisticadas. Devido a isso, muitas vezes é impossível desenvolver um perfil operacional confiável. Consequentemente, você não pode ter confiança sobre a exatidão de quaisquer medições de confiabilidade, pois elas podem ser baseadas em suposições incorretas sobre as maneiras como o sistema é usado.

Figura 15.3 Um perfil operacional

15.3 Testes de proteção

A avaliação da proteção de sistema é cada vez mais importante, pois sistemas mais críticos são permitidos na Internet e, assim, podem ser acessados por qualquer pessoa com uma conexão de rede. Existem histórias diárias sobre os ataques a sistemas baseados em Web, e os vírus e *worms* são distribuídos regularmente usando-se protocolos da Internet.

Tudo isso significa que os processos de verificação e validação de sistemas baseados em Web devem centrar-se na avaliação de proteção, em que é testada a capacidade do sistema de resistir a diferentes tipos de ataque. No entanto, como explica Anderson (2001), esse tipo de avaliação de proteção é muito difícil de se realizar. Como consequência, frequentemente, os sistemas são implantados com brechas de proteção. Os invasores aproveitam-se delas para ganhar acesso ao sistema ou causar danos ao sistema ou seus dados.

Fundamentalmente, existem duas razões pelas quais os testes de proteção são tão difíceis:

1. Requisitos de proteção, como alguns requisitos de segurança, são requisitos do tipo 'não deve'. Isso quer dizer que especificam o que não deve acontecer, em vez de especificarem alguma funcionalidade de sistema ou comportamento requerido. Geralmente, não é possível definir esse comportamento não desejado como simples limitações a serem verificadas pelo sistema.

   Se os recursos estiverem disponíveis, você poderá demonstrar, no princípio, pelo menos, que um sistema cumpre seus requisitos funcionais. Entretanto, é impossível provar que um sistema não faz algo. Independentemente da quantidade de testes, algumas vulnerabilidades de proteção podem permanecer em um sistema depois de implantado. Naturalmente, você pode gerar requisitos funcionais projetados para proteger o sistema de alguns ataques conhecidos. Entretanto, você não pode derivar requisitos para ataques desconhecidos ou não antecipados. Mesmo nos sistemas que estiveram em uso por muitos anos, um invasor engenhoso pode descobrir uma nova forma de ataque e penetrar em um sistema planejado para ser protegido.

2. As pessoas que atacam um sistema são inteligentes e estão procurando ativamente vulnerabilidades que possam explorar. Estão dispostas a experimentar o sistema e tentar coisas diferentes daquelas de uso ou atividade normais do sistema. Por exemplo, em um campo de sobrenome podem entrar mil caracteres com uma mistura de letras, pontuação e números. No entanto, uma vez que uma vulnerabilidade é encontrada, eles podem trocar informações sobre isso e, dessa forma, aumentar o número de potenciais invasores.

Os invasores podem tentar descobrir as suposições feitas pelos desenvolvedores de sistema e contradizer essas suposições para ver o que acontece. Eles usam e exploram um sistema durante um período de tempo e analisam esse sistema com ferramentas de software para descobrir vulnerabilidades que possam explorar. De fato, eles podem ter mais tempo para gastar na busca por vulnerabilidades do que os engenheiros de teste de sistema, assim como testadores também devem concentrar-se nos testes de sistema.

Por essa razão, a análise estática pode ser particularmente útil como ferramenta de teste de proteção. Uma análise estática de um programa pode guiar a equipe de teste para as áreas de um programa em que possam incluir erros e vulnerabilidades. As anomalias reveladas na análise estática podem ser reparadas diretamente ou podem ajudar a identificar os testes que precisam ser feitos para revelar se essas anomalias realmente representam um risco para o sistema.

Para verificar a proteção de um sistema, você pode usar uma combinação de testes, de análises baseadas em ferramentas e de verificação formal:

1. *Testes baseados em experiência*. Nesse caso, o sistema é analisado contra os tipos de ataques conhecidos pela equipe de validação, o que pode envolver o desenvolvimento de casos de teste ou exames de código-fonte de um sistema. Por exemplo, para certificar-se de que o sistema não seja suscetível ao conhecido ataque de envenenamento de SQL, você pode testá-lo usando entradas que incluam comandos de SQL. Para certificar-se de que não ocorram os erros de *overflow* de *buffer*, você pode examinar todos os *buffers* de entrada para ver se o programa está verificando essas atribuições aos elementos de *buffer* que estejam dentro dos limites.

Geralmente, esse tipo de validação é feito em conjunto com validação baseada em ferramentas, em que as ferramentas dão informações que ajudam a manter o foco do teste de sistema. Para ajudar no processo, podem ser criados *checklists* de verificação de problemas conhecidos de proteção. O Quadro 15.1 dá alguns exemplos de perguntas que podem ser usadas para dirigir testes baseados em experiências. Verifica-se se as diretrizes de pro-

Quadro 15.1 Exemplos de entradas em um *checklist* de proteção

***Checklist* de proteção**

1. Todos os arquivos criados na aplicação têm permissões de acesso apropriadas? Quando erradas, as permissões de acesso podem levar ao acesso desses arquivos por usuários não autorizados.
2. O sistema automaticamente encerra as sessões de usuário depois de um período de inatividade? Sessões que ficam ativas podem permitir acesso não autorizado através de um computador não usado.
3. Se o sistema for escrito em uma linguagem de programação sem verificação de limites de vetor, existem situações em que o *overflow* de *buffer* pode ser explorado? O *overflow* de *buffer* pode permitir que invasores enviem e executem sequências de código para o sistema.
4. Se as senhas forem definidas, o sistema verifica se elas são 'fortes'? Senhas fortes consistem de pontos, números e letras misturados e não são entradas de dicionário normal. Elas são mais difíceis de quebrar do que senhas simples.
5. As entradas do ambiente do sistema são sempre verificadas por meio da comparação com uma especificação de entrada? O tratamento incorreto de entradas mal formadas é uma causa comum de vulnerabilidades de proteção.

jeto e de programação para a proteção (Capítulo 14) foram seguidas e se podem ser incluídas em um *checklist* de problema de proteção.

2. *Equipes tigre*. Essa é uma forma de testes baseada em experiências, em que é possível aproveitar a experiência de fora da equipe de desenvolvimento para testar um sistema de aplicação. Você configura uma 'equipe tigre', à qual é dado o objetivo de violar a proteção de sistema. Eles simulam ataques ao sistema e usam criatividade para descobrir novas maneiras de comprometer a proteção do sistema. Membros de equipes tigre devem ter experiências anteriores com testes de proteção e em encontrar pontos fracos de proteção em sistemas.
3. *Testes baseados em ferramentas*. Nesse método, várias ferramentas de proteção, como verificadores de senha, são usadas para analisar o sistema. Os verificadores de senha detectam senhas inseguras, como nomes comuns ou sequências de letras consecutivas. Essa abordagem é a extensão da validação baseada em experiências, em que a experiência em falhas de proteção é incorporada nas ferramentas usadas. A análise estática é, naturalmente, outro tipo de teste baseado em ferramentas.
4. *Verificação formal*. Um sistema pode ser verificado contra uma especificação formal de proteção. No entanto, como em outras áreas, a verificação formal para proteção não é amplamente usada.

Inevitavelmente, os testes de proteção são limitados pelo tempo e recursos disponíveis para a equipe de teste. Isso significa que, normalmente, para testes de sistema, você deve adotar uma abordagem baseada em riscos e concentrar-se no que você acha que são os riscos mais significativos enfrentados pelo sistema. Se você tem uma análise dos riscos de proteção de sistema, ela poderá ser usada para conduzir o processo de teste. Assim como testar o sistema com os requisitos de segurança derivados desses riscos, a equipe de teste também deve tentar quebrar o sistema por meio da adoção de abordagens alternativas que ameacem os ativos do sistema.

É muito difícil para os usuários finais de um sistema verificarem sua proteção. Consequentemente, grupos de governos na América do Norte e na Europa estabeleceram conjuntos de critérios de avaliação de proteção que podem ser verificados por avaliadores especializados (PFLEEGER e PFLEEGER, 2007). Os fornecedores de produtos de software podem submeter seus produtos para avaliação e certificação de acordo com esses critérios. Consequentemente, se você tiver um requisito para um nível particular de proteção, poderá escolher um produto que seja validado nesse nível. Na prática, entretanto, esses critérios foram usados primeiramente em sistemas militares e, até o momento, não conseguiram muita aceitação comercial.

15.4 Garantia de processo

Conforme discutido no Capítulo 13, a experiência mostra que processos confiáveis conduzem a sistemas confiáveis. Isto é, se um processo for baseado em boas práticas de engenharia de software, então é mais provável que o produto de software resultante será confiável. Naturalmente, contudo, um bom processo não garante confiança. No entanto, evidências de que um processo confiável foi usado aumenta a confiança geral de que um sistema seja confiável. A garantia de processo está relacionada com a coleta de informações sobre os processos usados durante

o desenvolvimento de sistema, bem como os resultados desses processos. Essas informações fornecem evidências das análises, revisões e testes que foram feitos durante o desenvolvimento do software.

A garantia de processo está relacionada com duas coisas:

1. Temos os processos certos? Será que os processos de desenvolvimento de sistema usados na organização incluem controles e subprocessos de V & V adequados para o tipo de sistema a ser desenvolvido?
2. Estamos executando o processo corretamente? A organização realizou o trabalho de desenvolvimento tal como definido em suas descrições de processos de software e os resultados definidos no processo de software foram produzidos?

As empresas que possuem vasta experiência de engenharia de sistemas críticos têm evoluído seus processos para refletirem as boas práticas de verificação e validação. Em alguns casos, isso envolveu discussões com o regulador externo até um acordo sobre quais processos devem ser usados. Embora exista uma grande variação dos processos entre as empresas, as atividades que você deve encontrar nos processos de desenvolvimento de sistemas críticos incluem gerenciamento de requisitos, gerenciamento de mudanças e controle de configuração, modelagem de sistemas, revisões e inspeções, planejamento de testes e análise da cobertura de testes. A noção de melhoria de processos, em que boas práticas são introduzidas e institucionalizadas nos processos, é abordada no Capítulo 26.

Outro aspecto do processo de garantia é verificar se os processos foram devidamente aprovados, o que, normalmente, envolve garantir que os processos estejam devidamente documentados e verificar essa documentação de processo. Por exemplo, parte de um processo confiável pode envolver inspecções formais de programa. A documentação de cada inspeção deve incluir *checklists* usados para conduzir a inspeção, uma lista de pessoas envolvidas, os problemas identificados durante a inspeção e as ações necessárias.

Portanto, demonstrar que foi usado um processo confiável envolve produzir várias evidências de documentos sobre o processo e o software que está sendo desenvolvido. A necessidade dessa documentação extensa indica que processos ágeis são raramente usados em sistemas nos quais a certificação de segurança ou de confiança é necessária. Os processos ágeis concentram-se no software em si e (com razão) argumentam que uma grande parte da documentação de processo nunca é usada depois de produzida. No entanto, você precisa criar evidências e atividades de documentação de processo quando as informações sobre o processo são usadas como parte de um caso de segurança ou de confiança de sistema.

15.4.1 Processos para garantir segurança

A maioria dos trabalhos sobre a garantia de processos foi feita na área de desenvolvimento de sistemas críticos de segurança. Existem dois motivos pelos quais é importante que um processo de desenvolvimento de sistemas críticos de segurança inclua processos de V & V voltados para a análise e garantia de segurança:

1. Em sistemas críticos, os acidentes são eventos raros, e pode ser praticamente impossível simulá-los durante os testes. Não se pode confiar em testes abrangentes para replicar condições que possam gerar um acidente.
2. Requisitos de segurança, conforme discutido no Capítulo 12, são, por vezes, requisitos do tipo 'não deve' que excluem os comportamentos inseguros do sistema. É impossível demonstrar, conclusivamente, por meio de testes e outras atividades de validação, que esses requisitos foram atendidos.

As atividades específicas de garantia de segurança devem ser incluídas em todos os estágios do processo de desenvolvimento de software. Essas atividades de garantia de segurança registram as análises efetuadas e a pessoa ou pessoas responsáveis por essas análises. As atividades de garantia de segurança incorporadas nos processos de software podem incluir:

1. *Monitoramento e registro de perigos*, o que rastreia os perigos a partir de suas análises preliminares por meio da validação de testes de sistema.
2. *Revisões de segurança*, usadas em todo o processo de desenvolvimento.
3. *Certificação de segurança*, em que a segurança dos componentes críticos é formalmente certificada. Isso envolve um grupo externo para a equipe de desenvolvimento de sistema, examinando as evidências disponíveis e decidindo quando um sistema ou componente deve ser considerado seguro antes de ser disponibilizado para uso.

Para apoiar esses processos de garantia de segurança, devem ser nomeados os engenheiros de segurança de projeto que tenham explícita responsabilidade sobre os aspectos de segurança de um sistema. Isso significa que es-

sas pessoas serão responsabilizadas caso ocorra uma falha de sistema relacionada com a segurança. Esses profissionais devem ser capazes de demonstrar que as atividades de garantia de segurança foram executadas corretamente.

Os engenheiros de segurança trabalham com os gerentes de qualidade para garantir que um sistema detalhado de gerenciamento de configuração seja usado para rastrear todos os documentos relacionados com a segurança e mantê-los junto com a documentação técnica associada. Isso é essencial em todos os processos confiáveis. Existe pouco sentido em ter procedimentos rigorosos de validação, se uma falha no gerenciamento de configuração significa que o sistema errado será entregue ao cliente. Nos capítulos 24 e 25 são abordados os gerenciamentos de configuração e de qualidade.

O processo de análise de perigos, parte essencial do desenvolvimento de sistemas críticos de segurança, é um exemplo de um processo de garantia de segurança. A análise de perigos diz respeito à identificação dos perigos, sua probabilidade de ocorrência e a probabilidade de cada perigo gerar um acidente. Se existem códigos de programa que verificam e lidam com cada perigo, você pode argumentar que esses perigos não resultarão em acidentes. Tais argumentos podem ser suplementados por argumentos de segurança, conforme discutido posteriormente neste capítulo. Em casos nos quais a certificação externa é requerida antes de o sistema ser usado (por exemplo, em um avião), é geralmente uma condição para a certificação que a rastreabilidade possa ser demonstrada.

O documento central de segurança que deve ser produzido é o *log* de perigos. Esse documento fornece evidências que provam como os perigos identificados foram levados em conta durante o desenvolvimento de software. Esse *log* de perigos é usado em cada estágio do processo de desenvolvimento de software para documentar como cada estágio de desenvolvimento tratou os perigos. Um exemplo simplificado de uma entrada de *log* de perigos para o sistema de fornecimento de insulina é mostrado no Tabela 15.2. Esse formulário documenta o processo da análise de perigos e mostra os requisitos de projeto gerados durante esse processo. Esses requisitos de projeto destinam-se a garantir que o sistema de controle nunca possa entregar uma overdose de insulina para um usuário da bomba de insulina.

Tabela 15.2 Uma entrada de *log* de perigos simplificada

Log de perigos	Página 4: Impresso 20.02.2009			
Sistema: sistema de bomba de insulina	Arquivo:InsulinPump/Segurança/HazardLog			
Engenheiro de segurança: James Brown	Versão de *Log*: 1/3			
Perigo identificado	Overdose de insulina fornecida ao paciente			
Identificado por	Jane Williams			
Classe de criticidade	1			
Risco identificado	Alto			
Árvore de defeitos identificada	Sim	Data 24.01.07	Local	*Log* de perigos, Página 5
Criadores de árvore de defeitos	Jane Williams e Bill Smith			
Árvore de defeitos verificada	Sim	Data 17.01.07	Verificador	James Brown
Requisitos de projeto de segurança de sistema				
1. O sistema deve incluir software de autoteste, o qual testará o sistema de sensor, o relógio e o sistema de fornecimento de insulina.				
2. O software de autoverificação deve ser executado uma vez por minuto.				
3. No caso do software de autoverificação descobrir um defeito em qualquer um dos componentes de sistema, será emitido um aviso sonoro e o *display* da bomba deve indicar o nome do componente em que o defeito foi descoberto. O fornecimento de insulina será suspenso.				
4. O sistema deve incorporar um sistema de configuração que permite que o usuário do sistema modifique a dose de insulina computada que será fornecida pelo sistema.				
5. A quantidade de configuração não deve ser maior que um valor preestabelecido (maxOverride), definido quando o sistema é configurado pela equipe médica.				

Como mostrado na Tabela 15.2, os indivíduos que têm responsabilidades de segurança devem ser explicitamente identificados. Isso é importante por duas razões:

1. Quando as pessoas são identificadas, elas podem ser responsabilizadas por suas ações. Isso significa que elas tendem a ser mais cuidadosas, pois os problemas, quando rastreados, podem prejudicar seu trabalho.
2. Em casos de acidente, pode haver processos judiciais ou um inquérito. É importante ser capaz de identificar quem era responsável pela garantia de segurança para que essas pessoas possam responder por suas ações.

15.5 Casos de segurança e confiança

Os processos de verificação de segurança e de confiabilidade geram uma grande quantidade de informações, as quais podem incluir resultados de testes, informações sobre os processos de desenvolvimento, registros de reuniões de revisão etc. Essas informações fornecem evidências sobre a proteção e a confiança de um sistema e são usadas para ajudar a decidir se esse é ou não um sistema confiável o suficiente para ser usado operacionalmente.

Casos de segurança e confiança são documentos estruturados que estabelecem argumentos e evidências detalhados de que um sistema é seguro ou que se alcançou o nível exigido de proteção ou confiança. Algumas vezes, são chamados de casos de garantia. Essencialmente, os casos de segurança ou confiança unem todas as provas que demonstram que um sistema é confiável. Para muitos tipos de sistema críticos, a produção de um caso de segurança é um requisito legal. O caso deve satisfazer um órgão regulador ou de certificação antes de o sistema ser implantado.

A responsabilidade de um órgão regulador é verificar se um sistema completo é tão seguro ou confiável quanto possível, assim que seu papel ganha importância, principalmente quando um projeto de desenvolvimento está concluído. No entanto, reguladores e desenvolvedores raramente trabalham isoladamente; eles se comunicam com a equipe de desenvolvimento para estabelecer o que deve ser incluído para a segurança. O regulador e os desenvolvedores analisam os processos e procedimentos em conjunto para se certificar de que estes serão aprovados e documentados para satisfação do regulador.

Normalmente, os casos de confiança são desenvolvidos durante e após o processo de desenvolvimento de sistema. Às vezes, isso pode causar problemas se as atividades do processo de desenvolvimento não produzirem evidências de confiança do sistema. Graydon et al. (2007) argumentam que o desenvolvimento de um caso de segurança e confiança deve ser totalmente integrado com o projeto e implementação de sistema. Isso significa que as decisões de projeto de sistema podem ser influenciadas pelos requisitos dos casos de confiança. Devem ser evitadas as opções de projeto que possam aumentar significativamente as dificuldades e os custos de desenvolvimento de casos.

Casos de confiança são generalizações de casos de segurança de sistema. Um caso de segurança é um conjunto de documentos que inclui uma descrição do sistema a ser certificado, informações sobre os processos usados para desenvolver o sistema e, criticamente, argumentos lógicos que demonstrem que o sistema é suscetível de ser seguro. De uma forma mais concisa, Bishop e Bloomfield (1998) definem um caso de segurança como:

> *Um corpo de evidências documentado, que fornece argumentos convincentes e válidos de que um sistema é suficientemente seguro para determinada aplicação, em determinado ambiente.*

A organização e o conteúdo de um caso de segurança ou confiança dependem do tipo de sistema que será certificado e seu contexto operacional. A Tabela 15.3 mostra uma estrutura possível para um caso de segurança, mas não existem, nessa área, padrões industriais amplamente usados para casos de segurança. As estruturas do caso de segurança variam de acordo com a indústria e a maturidade do domínio. Por exemplo, casos de segurança nuclear foram requeridos por muitos anos. Eles são muito abrangentes e apresentados de maneira familiar para os engenheiros nucleares. No entanto, casos de segurança para dispositivos médicos foram introduzidos muito mais recentemente. Sua estrutura é mais flexível e os casos são menos detalhados que os casos nucleares.

Certamente, o software em si não é perigoso; apenas quando ele está embutido em um grande sistema baseado em computadores ou em sistemas sociotécnicos as falhas de software podem resultar em falhas de outros equipamentos ou processos que podem causar ferimentos ou mortes. Portanto, um caso de segurança de software é sempre parte de um caso de segurança de sistema mais amplo que demonstra a segurança do sistema global. Ao construir um caso de segurança de software, você deve relacionar as falhas de software com falhas de sistema globais e demonstrar que essas falhas de software não ocorrerão ou não serão propagadas de forma que possam ocorrer falhas perigosas de sistema.

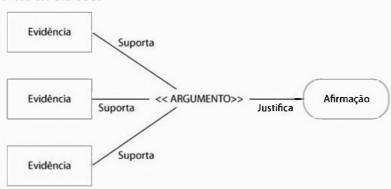
Tabela 15.3 O conteúdo de um caso de segurança de software

Capítulo	Descrição
Descrição de sistema	Uma visão geral do sistema e uma descrição de seus componentes essenciais.
Requisitos de segurança	Os requisitos de segurança abstraídos da especificação de requisitos de sistema. Detalhes de outros requisitos relevantes de sistema também podem ser incluídos.
Análise de perigos e riscos	Documentos que descrevem os perigos e os riscos identificados e as medidas tomadas para reduzir os riscos. Análises de risco e *logs* de perigos.
Análise de projeto	Um conjunto de argumentos estruturados (ver Seção 15.5.1) que justifique por que o projeto é seguro.
Verificação e validação	Uma descrição dos procedimentos de V & V usados e, se for o caso, os planos de teste para o sistema. Resumos dos resultados de testes mostrando defeitos que foram detectados e corrigidos. Se foram usados métodos formais, uma especificação formal de sistema e quaisquer análises dessa especificação. Registros de análises estáticas do código-fonte.
Relatórios de revisão	Registros de todas as revisões de projeto e segurança.
Competências de equipe	Evidência da competência de todos da equipe envolvida no desenvolvimento e validação de sistemas relacionados com a segurança.
Processo de QA	Registros dos processos de garantia de qualidade (ver Capítulo 24) efetuados durante o desenvolvimento de sistema.
Processos de gerenciamento de mudanças	Registros de todas as mudanças propostas, ações tomadas e, se for o caso, justificativa da segurança dessas mudanças. Informações sobre os procedimentos de gerenciamento de configuração e *logs* de gerenciamento de configuração.
Casos de segurança associados	Referências a outros casos de segurança que podem afetar o processo de segurança.

15.5.1 Argumentos estruturados

A decisão sobre se um sistema é suficientemente confiável para ser usado deve se basear em argumentos lógicos, os quais devem demonstrar que as provas apresentadas suportam as afirmações sobre a proteção e a confiança do sistema. Essas afirmações podem ser absolutas (evento X vai ou não vai acontecer) ou probabilísticas (a probabilidade de ocorrência do evento Y é 0, n). Um argumento liga a evidência e a afirmação. Como mostrado na Figura 15.4, um argumento é um relacionamento entre o que é pensado para ser o caso (a afirmação) e um corpo de evidências do que foi coletado. O argumento, essencialmente, explica por que a afirmação, que é uma asserção sobre a proteção de sistema ou confiança, pode ser inferida da evidência disponível.

Por exemplo, a bomba de insulina é um dispositivo crítico de segurança cuja falha pode causar prejuízos a um usuário. Em muitos países, isso significa que uma autoridade reguladora (no Reino Unido, a Direção de Dispositivos Médicos) precisa ser convencida da segurança do sistema antes que o dispositivo possa ser vendido e usado. Para

Figura 15.4 Argumentos estruturados

tomar essa decisão, o regulador avalia o caso de segurança para o sistema, que apresenta argumentos estruturados de que o funcionamento normal do sistema não causará danos a um usuário.

Geralmente, os casos de segurança dependem de argumentos estruturados, baseados em afirmações. Por exemplo, o seguinte argumento pode ser usado para justificar uma afirmação de que cálculos efetuados pelo software de controle não levarão a uma *overdose* de insulina, quando entregue a um usuário da bomba. Naturalmente, esse é um argumento muito simplificado. Em um caso real de segurança, seriam apresentadas referências mais detalhadas para as evidências.

Afirmação: A dose máxima única calculada pela bomba de insulina não excederá a maxDose, avaliada como uma única dose segura para um paciente em particular.

Evidência: Argumento de segurança para programa de controle de software da bomba de insulina. (Discuto os argumentos de segurança adiante, nesta seção.)

Evidência: Conjuntos de dados de teste para a bomba de insulina. Em 400 testes, o valor da currentDose foi corretamente computado e nunca excedeu a maxDose.

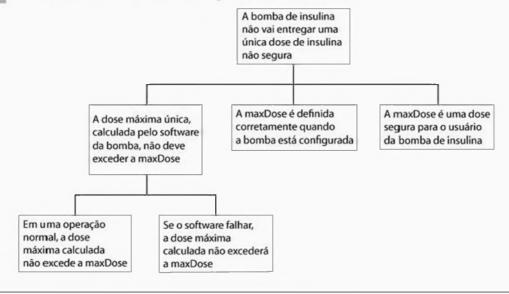
Evidência: O relatório de análise estática do programa de controle da bomba de insulina. A análise estática do software de controle revelou que nenhuma anomalia afetou o valor da currentDose, a variável de programa que contém a dose de insulina para ser entregue.

Argumento: A evidência apresentada mostra que a dose máxima de insulina que pode ser computada é igual à maxDose.

Assim, é razoável supor, com um elevado nível de confiança, que as evidências justificam a afirmação de que a bomba de insulina não calculará uma dose de insulina maior que a dose única máxima.

Observe que as eviências apresentadas são redundantes e diversificadas. O software é verificado por meio de vários mecanismos diferentes com sobreposições significativas entre eles. Como discutido no Capítulo 13, o uso de processos redundantes e diversificados aumenta a confiança. Se houver omissões e erros não detectados por um processo de validação, existe uma boa chance de eles serem encontrados por um dos outros processos.

Evidentemente, é normal que ocorram muitas afirmações sobre a confiança e a proteção de um sistema, com a validade de uma afirmação muitas vezes dependendo da validade de outras afirmações. Portanto, as afirmações podem ser organizadas em uma hierarquia. A Figura 15.5 mostra parte dessa hierarquia de afirmações para a bomba de insulina. Para demonstrar que uma afirmação de alto nível é válida, você primeiro tem de trabalhar com os argumentos para as afirmações de nível inferior. Se você puder mostrar que cada uma dessas afirmações de nível inferior é justificada, em seguida você poderá ser capaz de inferir que as afirmações de nível superior são justificadas.

Figura 15.5 Uma hierarquia de afirmações de segurança para a bomba de insulina

15.5.2 Argumentos estruturados de segurança

Argumentos estruturados de segurança são um tipo de argumento que demonstra que um programa cumpre suas obrigações de segurança. Em um argumento de segurança, não é necessário provar que o programa funciona conforme o esperado; só é necessário mostrar que a execução do programa não pode resultar em um estado inseguro. Isso significa que é mais econômico fazer argumentos de segurança do que argumentos de correção. Você não deve considerar todos os estados do programa — mas, simplesmente, concentrar-se em estados que poderiam causar um acidente.

Uma suposição geral que sustenta o trabalho em segurança de sistema é que o número de defeitos de sistema que podem levar a perigos críticos de segurança é significativamente menor do que o número total de defeitos que podem existir no sistema. A garantia de segurança pode concentrar-se nesses defeitos potencialmente perigosos. Se puder ser demonstrado que esses defeitos não podem ocorrer ou que, se ocorrerem, o perigo associado não resultará em um acidente, então o sistema é seguro. Essa é a base dos argumentos estruturados de segurança.

Os argumentos estruturados de segurança destinam-se a demonstrar que, assumindo condições normais de execução, um programa deve ser seguro. Geralmente, eles são baseados em contradição. As etapas envolvidas na criação de um argumento de segurança são as seguintes:

1. Você começa por assumir que um estado inseguro, que tenha sido identificado na análise de perigos de sistema, pode ser alcançado por meio da execução do programa.
2. Você escreve um predicado (uma expressão lógica) que defina esse estado inseguro.
3. Em seguida, você analisa sistematicamente um modelo de sistema ou o programa e mostra que todos os caminhos do programa levam até esse estado; a condição de terminação desses caminhos contradiz o predicado de estado inseguro. Se esse for o caso, a hipótese inicial de um estado inseguro está incorreta.
4. Após repetir essa análise para todos os perigos identificados, você terá forte evidência de que o sistema é seguro.

Os argumentos estruturados de segurança podem ser aplicados em diferentes níveis, desde os requisitos, por meio dos modelos de projeto até o código. No nível dos requisitos, você está tentando demonstrar que não estão faltando requisitos de segurança e que os requisitos não fazem suposições inválidas sobre o sistema. No nível de projeto, você pode analisar um modelo de estado do sistema para encontrar estados inseguros. No nível de código, você considera todos os caminhos por meio do código crítico de segurança para mostrar que a execução de todos os caminhos leva a uma contradição.

Por exemplo, considere o código no Quadro 15.2, que pode ser parte da implementação do sistema de fornecimento de insulina. O código calcula a dose de insulina a ser entregue, em seguida, aplicam-se algumas verificações de segurança para reduzir a probabilidade de uma *overdose*. O desenvolvimento de um argumento de segurança para esse código envolve demonstrar que a dose de insulina administrada nunca é maior que o nível máximo seguro para uma dose única. Isso é estabelecido para cada usuário diabético individualmente em reuniões com seus médicos.

Para demonstrar a segurança, não será necessário provar que o sistema entrega a dose 'correta'; apenas deve-se demonstrar que ele nunca entrega uma *overdose* para o paciente. Você deve trabalhar com o pressuposto de que a maxDose é um nível seguro para os usuários do sistema.

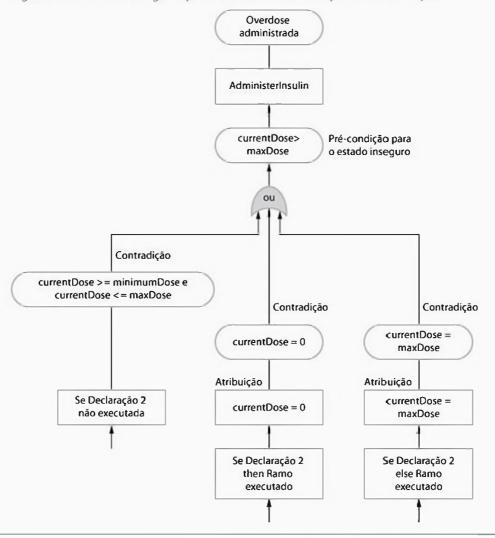
Para construir o argumento de segurança, você identifica o predicado que define o estado não seguro, que é currentDose > maxDose. Em seguida, demonstra que todos os caminhos do programa conduzem a uma contradição a essa asserção não segura. Se esse for o caso, a condição não segura não pode ser verdadeira. Se você pode fazer isso, pode ter certeza de que o programa calculará uma dose não segura de insulina. Você pode estruturar e apresentar os argumentos de segurança graficamente, como mostra a Figura 15.6.

Construir um argumento estruturado para um programa não produz um cálculo inseguro. Primeiro, você identifica todos os possíveis caminhos por meio do código que poderia levar ao estado potencialmente inseguro. A partir dele, você retrocede e considera a última atribuição para todas as variáveis de estado, em cada caminho para esse estado inseguro. Se você puder mostrar que nenhum dos valores dessas variáveis são inseguros, então você demonstra que sua suposição inicial (que a computação é insegura) está incorreta.

Trabalhar retrocedendo é importante porque significa que você pode ignorar todos os estados diferentes dos estados finais que levam à condição de saída para o código. Os valores anteriores não importam para a segurança do sistema. Nesse exemplo, tudo com que você precisa se preocupar é o conjunto de valores possíveis da

Quadro 15.2 Cálculo de dose de insulina com verificações de segurança

```
 - A dose de insulina a ser entregue é uma função de:
- nível de açúcar no sangue, a dose anterior entregue e
- o tempo de entrega da dose anterior
currentDose = computeInsulin ().
/ / Verificar/ajustar a currentDose de segurança, se necessário.
/ / if declaração 1
if (previousDose = = 0)
{
if (currentDose > maxDose/2)
CurrentDose = maxDose/2.
}
else
if (currentDose > (previousDose * 2))
currentDose = previousDose * 2.
/ / if declaração 2
if (currentDose < minimumDose)
currentDose = 0;
else if (currentDose > maxDose)
currentDose = maxDose;
administerInsulin (currentDose)
```

Figura 15.6 Argumento informal de segurança baseado em demonstrações de contradições

currentDose imediatamente antes que o método administerInsulin seja executado. Você pode ignorar processamentos, como a if-declaração 1 no Quadro 15.2, no argumento de segurança, porque seus resultados são sobrescritos em declarações posteriores do programa.

No argumento de segurança da Figura 15.6, existem três possíveis caminhos de programa que levam à chamada para o método administerInsulin. Você precisa demonstrar que a quantidade de insulina entregue nunca será superior à maxDose. Consideram-se todos os possíveis caminhos de programa para administerInsulin:

1. Nenhum ramo do if-declaração 2 é executado. Isso só pode acontecer se a currentDose for maior ou igual à minimumDose e menor ou igual à maxDose. Essa é a pós-condição — uma asserção que é verdadeira após a declaração ser executada.
2. O ramo *then* do if-declaração 2 é executado. Nesse caso, a atribuição de currentDose para zero é executada. Portanto, sua pós-condição é currentDose = 0.
3. O ramo *else if* do if-declaração 2 é executado. Nesse caso, a atribuição de currentDose para maxDose é executada. Portanto, depois que essa declaração tenha sido executada, saberemos que a pós-condição é currentDose = maxDose.

Em todos os três casos, as pós-condições contradizem a pré-condição insegura de a dose administrada ser maior que a maxDose. Podemos, portanto, afirmar que a computação é segura.

Da mesma forma, os argumentos estruturados podem ser usados para demonstrar que certas propriedades de segurança de um sistema são verdadeiras. Por exemplo, se você quer demonstrar que a computação nunca levará à permissão de um recurso que está sendo alterado, você pode ser capaz de usar um argumento estruturado de segurança para mostrar isso. No entanto, as evidências dos argumentos estruturados são menos confiáveis para a validação de segurança. Isso ocorre porque existe a possibilidade de o invasor corromper o código do sistema. Nesse caso, o código executado não é o código que você tem afirmado ser seguro.

PONTOS IMPORTANTES

- A análise estática é uma abordagem para a V & V que examina o código-fonte (ou outra representação) de um sistema à procura de erros e anomalias. Ela permite que todas as partes de um programa sejam verificadas, não apenas aquelas que são exercidas por testes de sistema.
- A verificação de modelos é uma abordagem formal para a análise estática que verifica exaustivamente todos os estados de um sistema, buscando possíveis erros.
- Os testes estatísticos são usados para se estimar a confiabilidade de software. Baseiam-se em testar o sistema com um conjunto de dados de teste que reflitam o perfil operacional do software. Os dados de teste podem ser gerados automaticamente.
- A validação de proteção é difícil porque os requisitos de proteção informam o que não deve acontecer em um sistema, e não o que deve. Além disso, os invasores de sistema são inteligentes e podem ter mais tempo para investigar os pontos fracos do que o tempo disponível para os testes de proteção.
- A validação de proteção pode ser efetuada com análises baseadas em experiências, com base em ferramentas ou 'equipes tigre' que simulam ataques a um sistema.
- É importante ter um processo certificado e bem definido para o desenvolvimento de sistemas críticos de segurança. O processo deve incluir a identificação e o controle de perigos potenciais.
- Os casos de segurança e confiança coletam todas as evidências que demonstram que um sistema é seguro e confiável. Os casos de segurança são necessários quando um regulador externo deve certificar o sistema antes de ele ser usado.
- Geralmente, os casos de segurança se baseiam em argumentos estruturados. Os argumentos estruturados de segurança mostram que uma condição de perigo identificado nunca pode ocorrer, consideraando todos os caminhos do programa que geram uma condição insegura e mostrando que a condição pode não se manter.

LEITURA COMPLEMENTAR

Software Reliability Engineering: More Reliable Software, Faster and Cheaper, 2nd edition. Provavelmente, esse é o livro definitivo sobre o uso de perfis operacionais e modelos de confiabilidade para avaliação de confiabilidade. Ele inclui detalhes de experiências com testes estatísticos. (MUSA, J. M. *Software Reliability Engineering*: More Reliable Software, Faster and Cheaper. McGraw-Hill, 2004).

'NASA's Mission Reliable'. Uma discussão sobre como a NASA usou a análise estática e verificação de modelos para garantir a confiabilidade do software de naves espaciais. (REGAN, P.; HAMILTON, S. *IEEE Computer*, v. 37, n. 1, jan. 2004.) Disponível em: <http://dx.doi.org/10.1109/MC.2004.1260727>.

'Dependability cases'. Uma introdução baseada em exemplos à definição de um caso de confiança. (WEINSTOCK, c. B.; GOODENOUGH, J. B.; HUDAK, J. J. *Software Engineering Institute*, CMU/SEI-2004-TN-016, 2004.) Disponível em: <http://www.sei.cmu.edu/publications/documents/04.reports/04tn016.html>.

How to Break Web Software: Functional and Security Testing of Web Applications and Web Services. Um livro pequeno que oferece bons conselhos práticos sobre como executar testes de proteção em aplicações em rede. (ANDREWS, M.; WHITTAKER, J. R. *How to Break Web Software*: Functional and Security Testing of Web Applications and Web Services. Addison-Wesley, 2006.)

'Using static analysis to find bugs'. Esse artigo descreve o Findbugs, um analisador estático Java que usa técnicas simples para encontrar possíveis violações de proteção e erros em tempo de execução. (AYEWAH, N. et al., *IEEE Software*, v. 25, n. 5, set./out. 2008.) Disponível em: <http://dx.doi.org/10.1109/MS.2008.130>.

EXERCÍCIOS

15.1 Explique quando pode ser efetivo usar a especificação e verificação formais no desenvolvimento de sistemas de software críticos de segurança. Por que os engenheiros de sistemas críticos são contra o uso de métodos formais? Dê sua opinião.

15.2 Sugira uma lista de condições que poderiam ser detectadas por um analisador estático para Java, C++ ou qualquer outra linguagem de programação que você usar. Comente essa lista quando comparada com a lista fornecida na Tabela 15.1.

15.3 Explique por que é praticamente impossível validar as especificações de confiabilidade quando elas são expressas em termos de um número muito pequeno de falhas durante a vida útil total de um sistema.

15.4 Explique por que garantir a confiabilidade de sistema não é uma garantia de segurança de sistema.

15.5 Usando exemplos, explique por que testes de segurança são um processo muito difícil.

15.6 Sugira como você validaria um sistema de proteção de senhas para uma aplicação que você tenha desenvolvido. Explique a função de todas as ferramentas que você acredita serem úteis.

15.7 O MHC-PMS precisa ser protegido contra ataques que podem revelar informações confidenciais sobre o paciente. Alguns desses ataques foram discutidos no Capítulo 14. Usando essa informação, estenda o *checklist* do Quadro 15.1 para orientar os testadores do MHC-PMS.

15.8 Liste quatro tipos de sistemas que podem exigir casos de segurança de software e explique por que os casos de segurança são necessários.

15.9 O mecanismo de controle de bloqueio de portas em uma instalação de armazenamento de resíduos nucleares é projetado para operação segura. Ele garante que a entrada para o armazém só seja permitida quando os escudos de radiação estão no lugar ou quando o nível de radiação no ambiente cai abaixo de um determinado valor (dangerLevel). Assim:

i. Se os escudos de radição remotamente controlados estiverem no lugar dentro de uma sala, um operador autorizado poderá abrir a porta.

ii. Se o nível de radiação em uma sala for inferior a um valor especificado, um operador autorizado poderá abrir a porta.

iii. Um operador autorizado será identificado pela entrada de um código autorizado de entrada de porta.

O código mostrado no Quadro 15.3 (a seguir) controla o mecanismo de bloqueio de porta. Note que o estado seguro é quando a entrada não deve ser permitida. Usando a abordagem discutida na Seção 15.5.2, desenvolva

Quadro 15.3 Código de entrada de porta

```
1   entryCode = lock.getEntryCode ().
2   if (entryCode = = lock.authorizedCode)
3   {
4       shieldStatus = Shield.getStatus ().
5       radiationLevel = RadSensor.get ().
6       if (radiationLevel < dangerLevel)
7           estado = seguro;
8       else
9           estado = inseguro;.
10      if (shieldStatus = = Shield.inPlace())
11          estado = seguro;
12      if (estado = = seguro)
13          {
14              Door.locked = false;
15              Door.unlock ().
16          }
17      else
18      {
19          Door.lock ( );
20          Door.locked : = true;
21      }
22  }
```

um argumento de segurança para esse código. Use os números de linha para se referir às declarações específicas. Se você acredita que o código é inseguro, sugira como ele deve ser modificado para que se torne seguro.

15.10 Considere que você era parte de uma equipe que desenvolveu o software para uma planta de produtos químicos que falhou, causando um grave incidente de poluição. Sua chefe é entrevistada na televisão e declara que o processo de validação é abrangente e que não existem defeitos no software. Ela afirma que os problemas devem ser devidos a procedimentos operacionais de porta. Um jornal aborda você e pede sua opinião. Discuta como você deve lidar com tal entrevista.

REFERÊNCIAS

ABRIAL, J. R. *The B Book*: Assigning Programs to Meanings. Cambridge, Reino Unido: Cambridge University Press, 2005.

ANDERSON, R. *Security Engineering*: A Guide to Building Dependable Distributed Systems. Chichester, Reino Unido: John Wiley & Sons, 2001.

BAIER, C.; KATOEN, J.-P. *Principles of Model Checking*. Cambridge, Mass.: MIT Press, 2008.

BALL, T.; BOUNIMOVA, E.; COOK, B.; LEVIN, V.; LICHTENBERG, J.; McGARVEY, C. et al. Thorough Static Analysis of Device Drivers. *Proc. EuroSys 2006*. Leuven, Bélgica, 2006.

BISHOP, P.; BLOOMFIELD, R. E. A methodology for safety case development. *Proc. Safety-critical Systems Symposium*. Birmingham, Reino Unido: Springer, 2008.

CHANDRA, S.; GODEFROID, P.; PALM, C. Software model checking in practice: An industrial case study. *Proc. 24th Int. Conf. on Software Eng.* (ICSE 2002), Orlando, Fia.: IEEE Computer Society, 2002. p. 431-441.

CROXFORD, M.; SUTTON, J. Breaking Through the V and V Bottleneck. *Proc. 2nda Int. Eurospace — Ada-Europe Symposium on Ada in Europe*. Frankfurt, Alemanha: Springer-LNCS, 2006. p. 344-354.

EVANS, D.; LAROCHELLE, D. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, v. 19, n. 1, 2002, p. 42-51.

GRAYDON, P. J.; KNIGHT, J. C.; STRUNK, E. A. Assurance Based Development of Critical Systems. *Proc. 37th Annual IEEE Conf. on Dependable Systems and Networks*. Edinburgh, Scotland, 2007, p. 347-357.

HOLZMANN, G. J. *The SPIN Model Checker*. Boston: Addison-Wesley, 2003.

KNIGHT, J. C.; LEVESON, N. G. Should software engineers be licensed? *Comm. ACM*, v. 45, n. 11, 2002, p. 87-90.

LARUS, J. R.; BALL, T.; DAS, M.; DELINE, R.; FAHNDRICH, M.; PINCUS, J. et al. Righting Software. *IEEE Software*, v. 21, n. 3, 2003, p. 92-100.

MUSA, J. D. *Software Reliability Engineering*: More Reliable Software, Faster Development and Testing. Nova York: McGraw-Hill, 1998.

NGUYEN, T.; OURGHANLIAN, A. Dependability assessment of safety-critical system software by static analysis methods. *Proc. IEEE Conf. on Dependable Systems and Networks (DSN'2003)*. San Francisco, Calif.: IEEE Computer Society, 2003, p. 75-79.

PFLEEGER, C. P.; PFLEEGER, S. L. *Security in Computing*. 4. ed. Boston: Addison-Wesley, 2007.

PROWELL, S. J.; TRAMMELL, C. J.; LINGER, R. C.; POORE, J. H. *Cleanroom Software Engineering*: Technology and Process. Reading, Mass.: Addison-Wesley, 1999.

REGAN, P.; HAMILTON, S. NASA's Mission Reliable. *IEEE Computer*, v. 37, n. 1, 2004, p. 59-68.

SCHNEIDER, S. *Concurrent and Real-time Systems*: The CSP Approach. Chichester, Reino Unido: John Wiley and Sons, 1999.

VISSER, W.; HAVELUND, K.; BRAT, G.; PARK, S.; LERDA, F. Model Checking Programs. *Automated Software Engineering J.*, v. 10, n. 2, 2003, p. 203-232.

VOAS, J. Fault Injection for the Masses. *IEEE Computer*, v. 30, n. 12, 1997, p. 129-130.

WORDSWORTH, J. *Software Engineering with B*. Wokingham: Addison-Wesley, 1996.

ZHENG, J.; WILLIAMS, L.; NAGAPPAN, N.; SNIPES, W.; HUDEPOHL, J. P.; VOUK, M. A. On the value of static analysis for fault detection in software. *IEEE Trans. on Software Eng.*, v. 32, n. 4, 2006, p. 240-245.

PARTE

1 2 **3** 4

Engenharia de software avançada

Esta parte do livro é intitulada 'Engenharia de software avançada' porque você precisa entender as bases da disciplina, discutidas nos capítulos 1 ao 9, para se beneficiar do material aqui apresentado. Muitos dos tópicos discutidos aqui refletem práticas de engenharia de software industrial no desenvolvimento de sistemas distribuídos e de tempo real.

O reúso de software tornou-se o paradigma de desenvolvimento dominante para sistemas de informação baseados na Web e sistemas corporativos. A abordagem mais comum para reusar é o reúso de COTS, em que um grande sistema é configurado para as necessidades de uma organização, e pouco ou nenhum desenvolvimento de software original é necessário. Apresento o tema geral de reúso no Capítulo 16 e, nesse capítulo, centro a discussão no reúso de sistemas COTS.

O Capítulo 17 também vai analisar o reúso de componentes de software em vez de sistemas de software como um todo. A engenharia de software baseada em componentes é um processo de composição de componentes com o novo código sendo desenvolvido para integrar componentes reusáveis. Nesse capítulo, explico o que se entende por um componente e por que os modelos de componentes-padrão são necessários para o reúso eficaz de componentes. Também discuto o processo geral da engenharia de software baseada em componentes e os problemas de composição de componentes.

Atualmente, a maioria dos grandes sistemas é de sistemas distribuídos. O Capítulo 18 trata de questões e problemas da construção de sistemas distribuídos. Vou apresentar a abordagem cliente-servidor como um paradigma fundamental da engenharia de sistemas distribuídos e explicar várias maneiras de implementar esse estilo de arquitetura. A seção final desse capítulo explica como o fornecimento de softwares como um serviço de aplicação distribuído mudará radicalmente o mercado de produtos de software.

O Capítulo 19 apresenta o tópico relacionado a arquiteturas orientadas a serviços, que ligam as noções de distribuição e reúso. Os serviços são componentes de software reusáveis, cuja funcionalidade pode ser acessada pela Internet e disponibilizada para uma gama de clientes. Nesse capítulo, explico o que está envolvido na criação de serviços (engenharia de serviços) e na composição de serviços para criação de novos sistemas de software.

Os sistemas embutidos são as instâncias de sistemas de software mais amplamente utilizadas. No Capítulo 20, eu trato desse importante tema. Apresento a ideia de um sistema embutido de tempo real e descrevo três padrões arquiteturais que são usados em projetos de sistemas embutidos. Em seguida, explico o processo de análise de *timing* e finalizo o capítulo com uma discussão sobre os sistemas operacionais de tempo real.

Finalmente, o Capítulo 21 abrange o desevolvimento de software orientado a aspectos (AOSD, do inglês *aspect-oriented software development*). O AOSD também está relacionado com o reúso e propõe uma nova abordagem, com base em aspectos, para organização e estruturação de sistemas de software. Embora não seja ainda a principal corrente na engenharia de software, o AOSD tem o potencial para melhorar significativamente nossas atuais abordagens de implementação de software.

Reúso de software

Objetivos

Os objetivos deste capítulo são apresentar o reúso de software e descrever abordagens para o desenvolvimento baseado em reúso de sistemas em grande escala. Com a leitura deste capítulo, você:

- entenderá os benefícios e problemas de reúso de software durante o desenvolvimento de novos sistemas;
- entenderá o conceito de um *framework* de aplicações como um conjunto de objetos reusáveis e como *frameworks* podem ser usados no desenvolvimento de aplicações;
- conhecerá as linhas de produtos de software que são compostas de uma arquitetura de núcleo comum e componentes configuráveis e reusáveis;
- aprenderá como sistemas podem ser desenvolvidos pela configuração e composição de sistemas de software de aplicação de prateleira.

Conteúdo

A engenharia de software baseada em reúso é uma estratégia da engenharia de software em que o processo de desenvolvimento é orientado para o reúso de softwares existentes. Apesar de o reúso ter sido proposto como uma estratégia de desenvolvimento há mais de 40 anos (McILROY, 1968), só em 2000 o 'desenvolvimento com reúso' se tornou a norma para novos sistemas de negócios. A mudança para o desenvolvimento baseado em reúso foi uma resposta às exigências de menores custos de produção e manutenção de software, entregas mais rápidas de sistemas e softwares de maior qualidade. Cada vez mais empresas consideram o software como um ativo valioso. O reúso tem sido promovido para aumentar o retorno sobre os investimentos em software.

A disponibilidade de softwares reusáveis tem aumentado significativamente. O movimento *open source* significa que existe uma enorme base de código reusável disponível a baixos custos. Isso pode dar-se na forma de bibliotecas de programas ou aplicações inteiras. Existem muitos sistemas de aplicação de domínios específicos disponíveis, os quais podem ser customizados e adaptados às necessidades de uma empresa específica. Algumas grandes empresas fornecem uma variedade de componentes reusáveis para seus clientes. Padrões, como os de *web service*, tornaram mais fácil o desenvolvimento de serviços gerais e reúso destes em uma variedade de aplicações.

A engenharia de software baseada em reúso é uma abordagem de desenvolvimento que tenta maximizar o reúso de softwares existentes. As unidades de software reusadas podem ser de tamanhos radicalmente diferentes. Por exemplo:

1. *Reúso de sistema de aplicação.* A totalidade de um sistema de aplicação pode ser reusada sem alterações em outros sistemas ou pela configuração da aplicação para diferentes clientes. Como alternativa, podem ser desenvolvidas

famílias de aplicações com uma arquitetura comum, mas adaptadas para clientes específicos. Ainda neste capítulo, eu trato do reúso de sistemas de aplicação.

2. *Reúso de componentes.* Os componentes de uma aplicação, variando em tamanho desde subsistemas até objetos únicos, podem ser reusados. Por exemplo, um sistema de identificação de padrões desenvolvido como parte de um sistema de processamento de textos pode ser reusado em um sistema de gerenciamento de banco de dados. Trato de reúso de componentes nos capítulos 17 e 19.
3. *Reúso de objetos e funções.* Componentes de software que implementam uma única função, como uma função matemática ou uma classe de objeto, podem ser reusados. Essa forma de reúso, baseada em bibliotecas-padrão, tem sido comum nos últimos 40 anos. Muitas bibliotecas de funções e classes estão disponíveis gratuitamente. Você reusa as classes e funções nessas bibliotecas, ligando-as com o código da aplicação recém-desenvolvido. Essa é uma abordagem particularmente eficaz em áreas como algoritmos e gráficos matemáticos, em que o conhecimento especializado é necessário para o desenvolvimento de funções e objetos eficientes.

Os componentes e os sistemas de software são entidades potencialmente reusáveis, mas, algumas vezes, sua natureza específica significa que é caro modificá-los para uma nova situação. Uma forma complementar de reúso é o 'reúso de conceito', em que, em vez de reusar um componente de software, você reusa uma ideia, uma forma, um trabalho ou um algoritmo. O conceito reusado é representado em uma notação abstrata (por exemplo, um modelo de sistema), que inclui detalhes de implementação. Pode, portanto, ser configurado e adaptado para uma série de situações. O conceito de reúso pode ser incorporado em abordagens como padrões de projeto (discutidos no Capítulo 7), produtos configuráveis de sistema e geradores de programa. Quando conceitos são reusados, o processo de reúso inclui atividades nas quais os conceitos abstratos são instanciados para criar componentes reusáveis executáveis.

Uma vantagem óbvia do reúso de software é a redução dos custos globais de desenvolvimento. Menos componentes de software precisam ser especificados, concebidos, implementados e validados. No entanto, a redução de custos é apenas uma das vantagens do reúso. Na Tabela 16.1, listei outras vantagens do reúso de ativos de software.

Contudo, há custos e problemas associados ao reúso (Tabela 16.2). Existe um custo significativo associado ao processo de compeender se, em determinada situação, um componente é adequado para o reúso, e em testar esse componente para garantia de sua confiança. Esses custos adicionais significam que as reduções nos custos de desenvolvimento por meio de reúso podem ser menores do que o previsto.

Tabela 16.1 Benefícios do reúso de software

Benefício	Explicação
Confiança aumentada	Os softwares reusados, experimentados e testados em sistemas em funcionamento provavelmente são mais confiáveis do que um novo software. Seus defeitos de projeto e implementação devem ser encontrados e corrigidos.
Risco de processo reduzido	O custo do software existente já é conhecido, considerando que os custos de desenvolvimento são sempre uma questão que envolve julgamento. Esse é um importante fator para o gerenciamento de projeto, pois reduz a margem de erro de estimativa de custos de projeto, o que é particularmente verdadeiro quando componentes de software relativamente grandes, como subsistemas, são reusados.
Uso eficaz de especialistas	Em vez de repetir o mesmo trabalho, especialistas em aplicações podem desenvolver softwares reusáveis que encapsulem seu conhecimento.
Conformidade com padrões	Alguns padrões, como os de interface de usuário, podem ser implementados como um conjunto de componentes reusáveis. Por exemplo, se os menus em uma interface de usuário forem implementados usando componentes reusáveis, todas as aplicações apresentarão os mesmos formatos de menu para os usuários. O uso de interfaces de usuário-padrão melhora a confiança, pois os usuários cometem menos erros quando são apresentados a interfaces familiares.
Desenvolvimento acelerado	Muitas vezes, os custos gerais de desenvolvimento não são tão importantes quanto entregar um sistema ao mercado o mais rápido possível. O reúso de um software pode acelerar a produção do sistema, pois pode reduzir o tempo de desenvolvimento e validação.

Tabela 16.2 Problemas com reúso

Problema	Explicação
Maiores custos de manutenção	Se o código-fonte de um sistema ou componentes de software reusáveis não estiverem disponíveis, os custos de manutenção podem ser maiores, pois os elementos reusados do sistema podem tornar-se cada vez mais incompatíveis com as alterações do sistema.
Falta de ferramentas de suporte	Algumas ferramentas de software não suportam o desenvolvimento com reúso. Pode ser difícil ou impossível integrar essas ferramentas com um sistema de biblioteca de componentes. O processo de software assumido por essas ferramentas pode não considerar o reúso. Isso é particularmente verdadeiro para as ferramentas que oferecem suporte a engenharia de sistemas embutidos, exceto para as ferramentas de desenvolvimento orientado a objetos.
Síndrome de 'não-inventado-aqui'	Alguns engenheiros de software preferem reescrever componentes, pois acreditam poder melhorá-los. Isso tem a ver, parcialmente, com aumentar a confiança e, parcialmente, com o fato de que escrever softwares originais é considerado mais desafiador do que reusar softwares de outras pessoas.
Criação, manutenção e uso de uma biblioteca de componentes	Preencher uma biblioteca de componentes reusáveis e garantir que desenvolvedores de software possam utilizar essa biblioteca podem ser ações caras. Processos de desenvolvimento precisam ser adaptados para garantir que a biblioteca seja usada.
Encontrar, compreender e adaptar os componentes reusáveis	Componentes de software precisam ser descobertos em uma biblioteca, compreendidos e, às vezes, adaptados para trabalhar em um novo ambiente. Os engenheiros precisam estar confiantes de que encontrarão, na biblioteca, um componente, antes de incluírem a pesquisa de componente como parte de seu processo normal de desenvolvimento.

Como discutido no Capítulo 2, os processos de desenvolvimento de software precisam ser adaptados para o reúso. Em particular, é necessário um estágio de refinamento dos requisitos em que os requisitos de sistema são modificados para refletir o software reusável que esteja disponível. Os estágios de projeto e implementação de sistema também podem incluir atividades explícitas para procurar e avaliar componentes candidatos ao reúso.

O reúso de software é mais eficaz quando está previsto como parte de um programa de reúso de toda a organização. Um programa de reúso envolve a criação de ativos reusáveis e a adaptação de processos de desenvolvimento para incorporar esses ativos no novo software. A importância do reúso de planejamento tem sido reconhecida por muitos anos no Japão (MATSUMOTO, 1984), onde o reúso é parte integrante da abordagem japonesa 'fábrica' para desenvolvimento de software (CUSAMANO, 1989). Empresas como a Hewlett-Packard também foram muito bem-sucedidas em seus programas de reúso (GRISS e WOSSER, 1995), e sua experiência foi documentada em um livro, por Jacobson e colegas. (1997).

16.1 O panorama de reúso

Ao longo dos últimos vinte anos, muitas técnicas foram desenvolvidas para oferecer suporte a reúso de software. Essas técnicas exploram os fatos de os sistemas, no mesmo domínio de aplicação, serem semelhantes e terem potencial para reúso. O reúso é possível em diferentes níveis, desde funções simples até aplicações completas, e normas para componentes reusáveis facilitam o reúso. A Figura 16.1 define várias possíveis maneiras de implementação de reúso de software. Cada uma será brevemente descrita na Tabela 16.3.

Dado esse conjunto de técnicas para reúso, a questão essencial é 'qual a técnica mais apropriada para se usar em uma determinada situação?'. Obviamente, isso depende dos requisitos para o sistema em desenvolvimento, a tecnologia e os ativos reusáveis disponíveis e a *expertise* da equipe de desenvolvimento. Fatores-chave que devem ser considerados ao planejar o reúso são:

1. *O cronograma de desenvolvimento para o software*. Caso o software necessite ser desenvolvido rapidamente, você deve tentar reusar sistemas de prateleira em vez de componentes individuais. Estes são ativos de alta granularidade reusáveis. Embora eles não se ajustem perfeitamente aos requisitos, essa abordagem minimiza a quantidade de desenvolvimento necessária.

Figura 16.1 O panorama de reúso

Tabela 16.3 Abordagens que apoiam o reúso de software

Abordagem	Descrição
Padrões de arquitetura	Padrões de arquitetura de software que oferecem suporte a tipos comuns de sistemas de aplicação são usados como base de aplicações. São descritos nos capítulos 6, 13 e 20.
Padrões de projeto	Abstrações genéricas que ocorrem em todas as aplicações são representadas como padrões de projeto, mostrando os objetos abstratos e concretos e as interações. São descritos no Capítulo 7.
Desenvolvimento baseado em componentes	Sistemas são desenvolvidos através da integração de componentes (coleções de objetos) que atendem aos padrões de modelos e componentes. São descritos no Capítulo 17.
Framework de aplicações	Coleções de classes abstratas e concretas são adaptadas e estendidas para criar sistemas de aplicação.
Empacotamento de sistemas legados	Sistemas legados (veja o Capítulo 9) são 'empacotados' pela definição de um conjunto de interfaces e acesso a esses sistemas legados por meio dessas interfaces.
Sistemas orientados a serviços	Sistemas são desenvolvidos pela ligação de serviços compartilhados, que podem ser fornecidos externamente. São descritos no Capítulo 19.
Linhas de produtos de software	Um tipo de aplicação é generalizado em torno de uma arquitetura comum para que esta possa ser adaptada para diferentes clientes.
Reúso de produto COTS	Sistemas são desenvolvidos pela configuração e integração de sistemas de aplicação existentes.
Sistemas de ERP	Sistemas de grande porte que sintetizam a funcionalidade e as regras de negócios genéricos são configurados para uma organização.
Aplicações verticais configuráveis	Sistemas genéricos são projetados para poder ser configurados para as necessidades dos clientes de sistemas específicos.
Bibliotecas de programas	Bibliotecas de classe e funções que implementam abstrações comumente usadas são disponibilizadas para reúso.
Engenharia dirigida a modelos	O software é representado como modelos de domínio e modelos de implementação independentes. O código é gerado a partir desses modelos. São descritos no Capítulo 5.
Geradores de programas	Um sistema gerador incorpora o conhecimento de um tipo de aplicação, e é usado para gerar sistemas nesse domínio a partir de um modelo de sistema fornecido pelo usuário.
Desenvolvimento de software orientado a aspectos	Quando o programa é compilado, os componentes compartilhados são integrados em uma aplicação em diferentes locais. São descritos no Capítulo 21.

2. *A expectativa de duração do software.* Caso você esteja desenvolvendo um sistema de vida longa, você deve centrar-se na manutenção de sistema. Você não deve apenas pensar nos benefícios imediatos do reúso, mas também nas implicações a longo prazo.

   Ao longo de sua vida, você terá de adaptar o sistema aos novos requisitos, o que significa fazer alterações em partes do sistema. Caso você não tenha acesso ao código-fonte, você pode preferir evitar componentes de prateleira e sistemas de fornecedores externos; fornecedores podem não ser capazes de continuar dando suporte ao software reusado.
3. *O conhecimento, as habilidades e a experiência do desenvolvimento da equipe.* Todas as tecnologias de reúso são bastante complexas, e é necessário muito tempo para compreendê-las e usá-las eficazmente. Portanto, se a equipe de desenvolvimento tem habilidades em uma área específica, provavelmente essa é a área em que você deve se concentrar.
4. *A importância do software e seus requisitos não funcionais.* Para um sistema crítico que precisa ser certificado por um regulador externo, talvez seja necessário criar um caso de confiança para o sistema (discutido no Capítulo 15). Isso é difícil, caso você não tenha acesso ao código-fonte do software. Se o software tiver requisitos de desempenho rigorosos, pode ser impossível usar estratégias como o reúso baseado em geradores, em que você gera o código a partir de uma representação específica de domínio reusável de um sistema. Esses sistemas geralmente geram códigos relativamente ineficientes.
5. *O domínio da aplicação.* Em alguns domínios de aplicação, como manufatura e sistemas médicos de informação, existem diversos produtos genéricos que podem ser reusados, sendo configurados a uma situação local. Se estiver trabalhando em tal domínio, você sempre deve considerar essa opção.
6. *A plataforma em que o sistema será executado.* Alguns modelos de componentes, como .NET, são específicos para plataformas Microsoft. Da mesma forma, sistemas de aplicação genéricos podem ser específicos de determinada plataforma e você só poderá reusá-los se seu sistema for projetado para a mesma plataforma.

A gama de técnicas de reúso disponível é tal que, na maioria das situações, existe a possibilidade de algum reúso de software. Se o reúso não é atingido, muitas vezes é uma questão de gerenciamento e não um problema técnico. Os gerentes podem estar dispostos a comprometer seus requisitos para permitir que componentes reusáveis sejam usados. Eles podem não compreender os riscos associados ao reúso tão bem quanto compreendem os riscos de desenvolvimento original. Embora os riscos de desenvolvimento de um novo software possam ser maiores, alguns gerentes podem preferir esses riscos por já serem conhecidos.

16.2 *Frameworks* de aplicações

Os entusiastas do desenvolvimento orientado a objetos sugerem que um dos benefícios principais de se usar uma abordagem orientada a objetos é que eles podem ser reusados em diferentes sistemas. No entanto, a experiência demonstra que, frequentemente, os objetos são muito pequenos e são especializados para uma aplicação específica. Leva mais tempo compreender e adaptar o objeto do que reimplementá-lo. Está claro que o reúso orientado a objetos é melhor suportado em um processo de desenvolvimento orientado a objetos por meio das abstrações de alta granularidade, chamadas *frameworks*.

Como o nome sugere, um *framework* é uma estrutura genérica estendida para se criar uma aplicação ou subsistema mais específico. Schmidt et al. (2004) definem um *framework* como:

> *"... um conjunto integrado de artefatos de software (como classes, objetos e componentes) que colaboram para fornecer uma arquitetura reusável para uma família de aplicações relacionadas."*

Os *frameworks* fornecem suporte para recursos genéricos, suscetíveis de serem usados em todas as aplicações de tipos semelhantes. Por exemplo, um *framework* de interface de usuário fornecerá suporte para tratamento de eventos de interface e incluirá um conjunto de recursos que possam ser usados para construir *displays*. Então, o desenvolvedor fica responsável por especializá-los, adicionando funcionalidade específica para uma aplicação específica. Por exemplo, em um *framework* de interface de usuário, o desenvolvedor define *layouts* de *display* apropriados para a aplicação que está sendo implementada.

Os *frameworks* dão suporte ao reúso de projeto, bem como ao reúso de classes específicas de sistema, pois fornecem uma arquitetura de esqueleto para a aplicação. A arquitetura é definida por classes de objetos e suas interações. As classes são reusadas diretamente e podem ser prorrogadas usando-se recursos, como a herança.

Os *frameworks* são implementados como uma coleção de classes de objetos concretos e abstratos em uma linguagem de programação orientada a objetos. Por conseguinte, *frameworks* são específicos da linguagem. Existem *frameworks* disponíveis em todas as linguagens de programação orientadas a objetos mais comumente usadas (por exemplo, Java, C# e C++, assim como linguagens dinâmicas como Ruby e Python). Na verdade, um *framework* pode incorporar vários outros, de forma que cada um deles é projetado para suportar o desenvolvimento de parte da aplicação. Você pode usar um *framework* para criar uma aplicação completa ou para implementar partes de uma aplicação, como a interface gráfica de usuário.

Fayad e Schmidt (1997) discutem três classes de *frameworks*:

1. Frameworks *de infraestrutura de sistema*. Esses *frameworks* apoiam o desenvolvimento de infraestruturas de sistema, como comunicações, interfaces de usuário e compiladores (SCHMIDT, 1997).
2. Frameworks *de integração de* middleware. Trata-se de um conjunto de normas e classes de objetos associados que oferecem suporte a componentes de comunicação e troca de informações. Exemplos desse tipo de *framework* incluem o .NET da Microsoft e o Enterprise Java Beans (EJB). Esses *frameworks* fornecem suporte para modelos de componentes padronizados, como discutirei no Capítulo 17.
3. Frameworks *de aplicações corporativas*. Esses estão relacionados com domínios de aplicação específicos, como telecomunicações ou sistemas financeiros (BAUMER et al., 1997). Eles incorporam conhecimentos sobre domínios de aplicações e apoiam o desenvolvimento de aplicações de usuário final.

Os *frameworks* de aplicações Web (WAFs, do inglês *web application frameworks*) são um tipo de *framework* mais recente e muito importante. WAFs que apoiam a construção de sites dinâmicos estão amplamente disponíveis. Geralmente, a arquitetura de um WAF é baseada no padrão Composite do MVC (Modelo-Visão-Controlador) (GAMMA et al., 1995), mostrado na Figura 16.2.

Originalmente, o padrão MVC foi proposto na década de 1980 como uma abordagem de projeto de GUI que permitiu várias apresentações de um objeto e estilos diferentes de interações com cada uma dessas apresentações. Ele permite a separação entre o estado da aplicação e a interface de usuário da aplicação. Um framework MVC oferece suporte à apresentação de dados de maneiras diferentes, e permite a interação com cada uma dessas apresentações. Quando os dados são modificados por meio de uma das apresentações, o modelo de sistema é alterado e os controladores associados a cada visão atualizam sua apresentação.

Muitas vezes, os *frameworks* são implementações de padrões de projeto, como discutido no Capítulo 7. Por exemplo, um *framework* MVC inclui o padrão Observer, o padrão Strategy, o padrão Composite e um número de outros que são discutidos por Gamma et al. (1995). A natureza geral dos padrões e o uso de classes abstratas e concretas permitem extensibilidade. Sem padrões, é quase certo que *frameworks* seriam impraticáveis.

Os WAFs normalmente incorporam um ou mais *frameworks* especializados, que oferecem suporte a recursos de aplicações específicos. Embora cada *framework* inclua uma funcionalidade ligeiramente distinta, a maioria dos WAFs suporta os seguintes recursos:

1. *Proteção*. WAFs podem incluir classes para ajudar a implementar a autenticação de usuário (*login*) e controle de acesso, para garantir que os usuários só possam acessar a funcionalidade permitida no sistema.

Figura 16.2 O padrão Modelo-Visão-Controlador

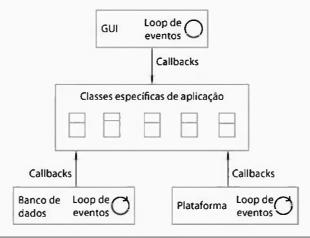
2. *Páginas Web dinâmicas*. As classes são fornecidas para ajudar na definição de *templates* de páginas Web e para preenchê-los dinamicamente, com dados específicos do banco de dados do sistema.
3. *Suporte de banco de dados*. Geralmente, os *frameworks* não incluem um banco de dados, mas assumem que um banco de dados separado, como MySQL, será usado. O quadro pode fornecer classes que proporcionam uma interface abstrata para bancos de dados diferentes.
4. *Gerenciamento de sessão*. Classes para criar e gerenciar sessões (um número de interações com o sistema por um usuário) são geralmente parte de um WAF.
5. *Interação de usuário*. Atualmente, a maioria dos *frameworks* Web fornece suporte a AJAX (HOLDENER, 2008), que permite que sejam criadas mais páginas Web interativas.

Para estender um *framework*, você não altera seu código. Em vez disso, você adiciona a ele classes concretas que herdam operações de classes abstratas. Além disso, talvez você precise definir *callbacks*. *Callbacks* são métodos chamados em resposta a eventos reconhecidos pelo *framework*. Schmidt et al. (2004) chamam esses métodos de 'inversão de controle'. Os responsáveis pelo controle no sistema são os objetos do *framework*, em vez de objetos específicos de aplicação. Em resposta a eventos de interface do usuário, banco de dados etc., esses objetos do *framework* invocam 'métodos *hook*' que, em seguida, são vinculados à funcionalidade fornecida ao usuário. A funcionalidade específica de aplicação responde ao evento de forma adequada (Figura 16.3). Por exemplo, um *framework* terá um método que lida com um clique de mouse a partir do ambiente. Esse método chama o método *hook*, que deve ser configurado para chamar os métodos de aplicação adequada para tratar o clique de mouse.

As aplicações que são construídas com *frameworks* podem ser a base para reúso futuro, por meio do conceito de linhas de produtos de software ou famílias de aplicações. Como essas aplicações são construídas por meio de um *framework*, modificar os membros da família para criar instâncias do sistema é, muitas vezes, um processo simples. Trata-se de reescrever classes concretas e métodos que você adicionou no *framework*.

No entanto, geralmente os *frameworks* são mais gerais que as linhas de produtos de software, que focam sobre uma família específica de sistema de aplicações. Por exemplo, você pode usar um *framework* baseado em Web para criar diferentes tipos de aplicações baseadas em Web. Um deles pode ser uma linha de produto de software que oferece suporte a *helpdesks* baseados em Web. Essa 'linha de produto de *helpdesk*' pode, em seguida, ser especializada para fornecer esses tipos específicos de suporte a *helpdesk*.

Os *frameworks* são uma abordagem eficaz de reúso, mas são caros para serem introduzidos em processos de desenvolvimento de software. Eles são inerentemente complexos e pode demorar meses para alguém aprender a usá-los. Pode ser difícil e caro avaliar *frameworks* disponíveis para a escolha do *framework* mais adequado. A depuração de aplicações baseadas em *framework* é difícil, pois você pode não compreender como os métodos de *framework* interagem. Esse é um problema geral dos softwares reusáveis; as ferramentas de depuração podem fornecer informações sobre os componentes de sistema reusados, as quais podem não ser compreendidas por um desenvolvedor.

Figura 16.3 Inversão de controle em *frameworks*

16.3 Linhas de produto de software

Uma das abordagens mais eficazes para o reúso é criar linhas de produto de software ou famílias de aplicações. Uma linha de produtos de software é um conjunto de aplicações com uma arquitetura comum e componentes compartilhados, sendo cada aplicação especializada para refletir necessidades diferentes. O núcleo do sistema é projetado para ser configurado e adaptado para atender às necessidades de clientes diferentes. Isso pode envolver a configuração de alguns componentes, implementação de componentes adicionais e a modificação de alguns dos componentes para refletir novos requisitos.

O desenvolvimento de aplicações pela adaptação de uma versão genérica da aplicação significa que uma alta proporção de códigos de aplicação é reusada. Além disso, a experiência de aplicação muitas vezes é transferível de um sistema para outro. Por conseguinte, quando os engenheiros de software participam de uma equipe de desenvolvimento, seu processo de aprendizagem é reduzido. Os testes são simples, pois testes para grande parte da aplicação também podem ser reusados, reduzindo, desse modo, o tempo de desenvolvimento geral da aplicação.

As linhas de produtos de software geralmente surgem de aplicações existentes. Isto é, uma organização desenvolve uma aplicação e, quando um sistema similar é requisitado, um sistema semelhante reúsa o código na nova aplicação, informalmente. O mesmo processo é usado quando outras aplicações similares são desenvolvidas. No entanto, a mudança tende a corromper a estrutura da aplicação. Como instâncias mais novas são desenvolvidas, é cada vez mais difícil a criação de uma nova versão. Consequentemente, pode-se tomar a decisão de se criar uma linha de produtos genéricos, o que envolve a identificação de funcionalidade comum nas instâncias dos produtos e a inclusão destas em uma aplicação de base, usada para desenvolvimento futuro. Essa aplicação de base é deliberadamente estruturada para simplificar o reúso e a reconfiguração.

Frameworks de aplicações e linhas de produtos de software têm, obviamente, muito em comum. Ambos oferecem suporte a uma arquitetura e componentes comuns e exigem um novo desenvolvimento para se criar uma versão específica de um sistema. As principais diferenças entre essas abordagens são:

1. *Frameworks* de aplicação dependem de recursos orientados a objetos como herança e polimorfismo para implementar as extensões para o *framework*. Geralmente, o código de *framework* não é modificado, e possíveis modificações limitam-se ao que é permitido pelo *framework*. Linhas de produtos de software não são necessariamente criadas usando-se uma abordagem orientada a objetos. Os componentes de aplicação são alterados, deletados ou rescritos. Em princípio, pelo menos, não há limites para as alterações que podem ser feitas.
2. *Frameworks* de aplicação concentram-se principalmente no apoio técnico e não de domínio específico. Por exemplo, existem *frameworks* de aplicações para se criar aplicações baseadas em Web. Uma linha de produtos de software geralmente incorpora informações detalhadas de domínio e de plataforma. Um exemplo poderia ser uma linha de produtos de software relacionada com aplicações baseadas em Web para o gerenciamento de registros de saúde.
3. Muitas vezes, as linhas de produtos de software são aplicações de controle para o equipamento. Por exemplo, pode haver uma linha de produtos de software para uma família de impressoras. Isso significa que a linha de produtos deve fornecer suporte para a interface de hardware. Geralmente, os *frameworks* de aplicação são orientados para software e raramente fornecem suporte para a interface de hardware.
4. Linhas de produtos de software são compostas de uma família de aplicações relacionadas, de propriedade da mesma organização. Quando você cria uma nova aplicação, o ponto de partida frequentemente é o membro mais próximo da família de aplicações, e não a aplicação central genérica.

Se você estiver desenvolvendo uma linha de produtos de software com uma linguagem de programação orientada a objetos, em seguida você pode usar um *framework* de aplicação como uma base para o sistema. Para criar o núcleo da linha de produtos, você pode estender o *framework* com componentes específicos de domínio usando seus mecanismos internos. Em seguida, há uma segunda fase de desenvolvimento, em que são criadas versões do sistema para diferentes clientes.

Podem ser desenvolvidos vários tipos de especialização de uma linha de produtos de software:

1. *Especialização de plataforma*. As versões da aplicação são desenvolvidas para diferentes plataformas. Por exemplo, versões da aplicação podem existir para plataformas Linux, Windows e Mac OS. Nesse caso, a funcionalidade da aplicação costuma ser inalterada. Apenas os componentes que fazem interface com o hardware e o sistema operacional são modificados.

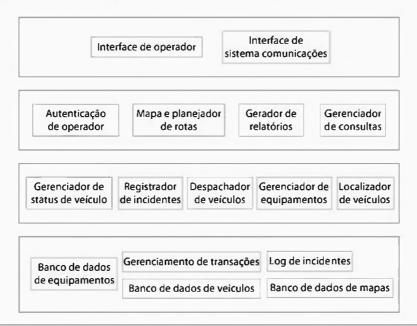
2. *Especialização de ambiente*. Versões da aplicação são criadas para lidar com ambientes operacionais específicos e dispositivos periféricos. Por exemplo, um sistema para os serviços de emergência pode existir em versões diferentes, dependendo do sistema de comunicações do veículo. Nesse caso, os componentes de sistema são alterados para refletir a funcionalidade do equipamento de comunicação usado.
3. *Especialização funcional*. Versões da aplicação são criadas para diferentes clientes, com requisitos específicos. Por exemplo, um sistema de automação de biblioteca pode ser modificado dependendo se ele é usado em uma biblioteca pública, uma biblioteca de referência ou uma biblioteca universitária. Nesse caso, os componentes que implementam a funcionalidade podem ser modificados, e novos componentes, adicionados ao sistema.
4. *Especialização de processo*. O sistema é adaptado para lidar com processos específicos de negócios. Por exemplo, um sistema de pedidos pode ser adaptado para lidar com um processo de solicitação centralizado, em uma empresa, e um processo distribuído, em outra.

Muitas vezes, a arquitetura de uma linha de produtos de software reflete um estilo ou padrão de arquitetura geral de aplicações específicas. Por exemplo, considere um sistema de linha de produto que é projetado para lidar com a expedição de veículos para serviços de emergência. Os operadores desse sistema recebem chamadas sobre incidentes, encontram o veículo adequado para responder ao incidente e despacham o veículo para o local do incidente. Os desenvolvedores de tal sistema podem negociar versões desse sistema para os serviços de polícia, bombeiros e ambulâncias.

Esse sistema de despacho de veículo é um exemplo de uma arquitetura de gerenciamento de recursos (Figura 16.4). Você pode ver como essa estrutura de quatro camadas é instanciada na Figura 16.5, que mostra os módulos que podem ser incluídos em uma linha de produtos de sistema de despacho de veículos. Os componentes de cada nível do sistema de linha de produtos são:

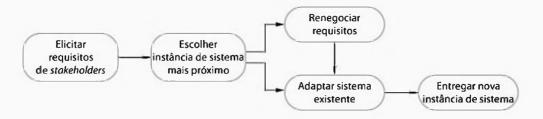
1. No nível de interação, existem componentes fornecendo uma interface de *display* do operador e uma interface com os sistemas de comunicações usados.
2. No nível de gerenciamento de E/S (nível 2), existem componentes que tratam a autenticação do operador, geram relatórios de incidentes e veículos despachados, suportam saídas de mapas e planejamento de rotas e fornecem um mecanismo para que os operadores consultem os bancos de dados do sistema.
3. No nível de gerenciamento de recursos (nível 3), existem componentes que permitem que veículos possam ser localizados e despachados, componentes para atualizar o *status* de veículos e equipamentos e um componente para registrar detalhes de incidentes.
4. No nível de banco de dados, bem como no suporte habitual ao gerenciamento de transações, existem bancos de dados separados de veículos, equipamentos e mapas.

Figura 16.4 A arquitetura de um sistema de alocação de recursos

Figura 16.5 A arquitetura de linha de produto de um sistema de despacho de veículos

Para criar uma versão específica do sistema, talvez você precise modificar componentes individuais. Por exemplo, a polícia tem um grande número de veículos, mas um pequeno número de tipos de veículos, considerando que o serviço de bombeiros tem muitos tipos de veículos especializados. Portanto, talvez você precise definir uma estrutura diferente para o banco de dados de veículos na implementação de um sistema para cada um desses serviços.

A Figura 16.6 mostra as etapas necessárias para estender uma linha de produtos de software para se criar uma nova aplicação. Geralmente, as etapas envolvidas nesse processo são:

1. *Elicitar requisitos de* stakeholders. Você pode começar com um processo normal de engenharia de requisitos. No entanto, como já existe um sistema, você precisará demonstrá-lo, e que os *stakeholders* o experimentem expressando suas necessidades como modificações nas funções fornecidas.
2. *Selecionar o sistema mais próximo aos requisitos.* Ao criar um novo membro de uma linha de produtos, você pode começar com a instância de produtos mais próxima. Os requisitos são analisados e o membro da família que é a opção mais próxima é escolhido para a modificação.
3. *Renegociar requisitos.* Assim que surgirem mais detalhes sobre as alterações necessárias e o projeto for planejado, podem ocorrer algumas renegociações de requisitos para minimizar as mudanças necessárias.
4. *Adaptar sistema existente.* Novos módulos são desenvolvidos para o sistema existente e módulos de sistemas existentes são adaptados para novos requisitos.
5. *Entregar novo membro da família.* A nova instância da linha de produtos é entregue ao cliente. Nessa etapa, você deve documentar suas principais características para que possam ser usadas como base para futuros desenvolvimentos de sistema.

Figura 16.6 Desenvolvimento de instância de produto

Quando você cria um membro da linha de produtos, é necessário encontrar um compromisso entre reusar, tanto quanto possível, a aplicação genérica e satisfazer os requisitos detalhados dos *stakeholders*. Quanto mais pormenorizados são os requisitos de sistema, é menos provável que existam componentes para atender a esses requisitos. No entanto, se os *stakeholders* estão dispostos a ser flexíveis e limitar as modificações do sistema, você provavelmente poderá entregar o sistema mais rapidamente e a baixo custo.

As linhas de produtos de software são projetadas para serem reconfiguradas, e essa reconfiguração pode envolver adicionar ou remover componentes do sistema, definir parâmetros e restrições para componentes de sistema e incluir o conhecimento de processos de negócios. Essa configuração pode ocorrer em diferentes estágios do processo de desenvolvimento:

1. *Configuração em tempo de projeto*. A organização que está desenvolvendo o software modifica um núcleo comum de linha de produto por desenvolvimento, seleção ou adaptação de componentes para criar um novo sistema para um cliente.
2. *Configuração em tempo de implantação*. Um sistema genérico é projetado por configuração para um cliente ou consultores, trabalhando com o cliente. O conhecimento dos requisitos específicos do cliente e do ambiente operacional do sistema é incorporado em um conjunto de arquivos de configuração que são usados pelo sistema genérico.

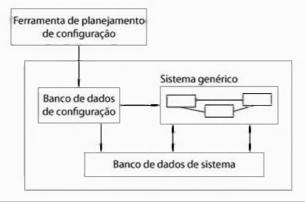
Quando um sistema é configurado em tempo de projeto, o fornecedor começa com um sistema genérico ou uma instância de um produto existente. Modificando e estendendo módulos nesse sistema, cria-se um sistema específico que oferece a funcionalidade necessária. Geralmente, esse processo envolve a alteração e extensão do código-fonte do sistema para uma maior flexibilidade, possivelmente com a configuração em tempo de implantação.

A configuração em tempo de implantação envolve o uso de uma ferramenta de configuração para criar uma configuração específica que é gravada em um banco de dados de configuração ou como um conjunto de arquivos de configuração (Figura 16.7). O sistema de execução consulta esse banco de dados durante a execução, de modo que sua funcionalidade possa ser especializada para seu contexto de execução.

Existem vários níveis de configuração em tempo de implantação que podem ser fornecidos em um sistema:

1. Seleção de componentes, em que você seleciona os módulos, em um sistema, que fornecem a funcionalidade necessária. Por exemplo, em um sistema de informações de pacientes, você pode selecionar um componente de gerenciamento de imagem que lhe permite vincular imagens médicas (radiografias, tomografias computadorizadas etc.) ao registro médico de pacientes.
2. Definição de *workflow* e regras, em que você define *workflows* (como a informação é processada, fase por fase) e a validação de regras que devem ser aplicadas às informações inseridas pelos usuários ou geradas pelo sistema.
3. Definição de parâmetro, em que você especifica os valores dos parâmetros específicos de sistema que refletem a instância da aplicação que você está criando. Por exemplo, você pode especificar o comprimento máximo dos campos de entrada de dados por um usuário ou as características de hardware conectadas ao sistema.

A configuração em tempo de implantação pode ser muito complexa; pode levar muitos meses para configurar o sistema para um cliente. Grandes sistemas configuráveis podem apoiar o processo de configuração, fornecendo ferramentas de software, tais como ferramentas de planejamento de configurações, para suportar o processo de configuração. Discuto a configuração em tempo de implantação adiante, na Seção 16.4.1. Ela abrange o reúso de sistemas COTS que precisam ser configurados para trabalharem em diferentes ambientes operacionais.

Figura 16.7 Configuração em tempo de implantação

A configuração em tempo de projeto é usada quando é impossível usar os recursos de configuração em tempo de implantação de um sistema já existente, para desenvolver uma nova versão de sistema. No entanto, ao longo do tempo, quando você tiver criado vários membros da família com funcionalidade comparável, você poderá decidir refatorar o núcleo da linha de produtos para incluir funcionalidade que foi implementada em vários membros da família de aplicação. Em seguida, quando o sistema for implantado, você pode fazer essa nova funcionalidade configurável.

16.4 Reúso de produtos COTS

Um produto de prateleira (COTS, do inglês *commercial-off-the-shelf*) é um sistema de software que pode ser adaptado às necessidades de diferentes clientes sem alterar o código-fonte do sistema. Praticamente todos os softwares de *desktop* e uma grande variedade de produtos de servidor são softwares COTS. Como esse software é projetado para uso geral, costuma incluir muitos recursos e funções. Portanto, tem o potencial de ser reusado em diferentes ambientes e como parte de aplicações diferentes. Torchiano e Morisio (2004) também descobriram que, muitas vezes, produtos *open source* foram usados como produtos COTS. Ou seja, os sistemas *open source* foram usados sem alteração e sem se olhar o código-fonte.

Os produtos COTS são adaptados por mecanismos internos de configuração que permitem que a funcionalidade do sistema seja customizada para se adaptar às necessidades específicas do cliente. Por exemplo, em um hospital, o sistema de registo de pacientes, formulários de entrada e relatórios de saída separados poderiam ser definidos para diferentes tipos de pacientes. Outros recursos de configuração podem permitir que o sistema aceite *plug-ins* que estendam a funcionalidade ou verifiquem entradas de usuários para garantir que estes sejam válidos.

Essa abordagem para o reúso de software tem sido amplamente aprovada por grandes empresas ao longo dos últimos 15 anos, pois oferece benefícios significativos para o desenvolvimento de software customizado:

1. Assim como em outros tipos de reúso, pode ser possível a implantação mais rápida de um sistema confiável.
2. É possível ver qual funcionalidade é fornecida pelas aplicações e, portanto, é mais fácil julgar se elas são suscetíveis de serem adequadas. Outras empresas podem usar as aplicações para disponibilizar a experiência dos sistemas.
3. Alguns riscos de desenvolvimento são evitados usando-se softwares existentes. No entanto, essa abordagem tem seus próprios riscos, como discuto adiante.
4. As empresas podem concentrar-se em sua atividade principal sem precisar dedicar uma grande quantidade de recursos para desenvolvimento de sistemas de TI.
5. Como as plataformas operacionais evoluem, as atualizações das tecnologias podem ser simplificadas, já que são de responsabilidade do fornecedor do produto COTS, e não do cliente.

Evidentemente, essa abordagem da engenharia de software tem seus próprios problemas:

1. Geralmente, os requisitos precisam ser adaptados para refletir a funcionalidade e podem gerar interrupções nos processos de negócios existentes.
2. O produto COTS pode basear-se em pressupostos praticamente impossíveis de se alterar. Portanto, o cliente deve adaptar seus negócios para refletir essas suposições.
3. A escolha do sistema COTS adequado para uma empresa pode ser um processo difícil, especialmente porque muitos produtos COTS não estão bem documentados. Fazer a escolha errada pode ser desastroso, pois pode ser impossível fazer o novo sistema trabalhar conforme requerido.
4. Pode haver falta de especialidade local para o desenvolvimento de sistemas de apoio. Por conseguinte, o cliente precisa contar com fornecedores e consultores externos para recomendação de desenvolvimento. Essa recomendação pode ser tendenciosa e orientada para a venda de produtos e serviços, em vez das reais necessidades do cliente.
5. O fornecedor do produto COTS controla o suporte e a evolução de sistema. Eles podem sair do negócio, podem ser comprados ou fazer alterações que causem dificuldades para os clientes.

O reúso de software baseado em COTS tornou-se cada vez mais comum. Atualmente, a maioria dos novos sistemas de processamento de informações de negócios é criada usando-se um COTS em vez de usar uma abordagem orientada a objetos. Embora muitas vezes existam problemas com essa abordagem para o desenvolvimento de sistema (TRACZ, 2001), histórias de sucesso (BAKER, 2002; BALK e KEDIA, 2000; BROWNSWORD e MORRIS, 2003; PFARR e REIS, 2002) mostram que o reúso baseado em COTS reduz o tempo e o esforço para implantação do sistema.

Existem dois tipos de reúso de produtos COTS, chamados sistemas de solução COTS e sistemas integrados COTS. Sistemas de solução COTS consistem em uma aplicação genérica, de um único fornecedor, configurada para os requisitos do cliente. Os sistemas integrados COTS envolvem a integração de dois ou mais sistemas COTS (talvez de diferentes fornecedores) para se criar um sistema de aplicação. A Tabela 16.4 resume as diferenças entre essas diferentes abordagens.

16.4.1 Sistemas de solução COTS

Sistemas de solução COTS são sistemas genéricos de aplicação que podem ser projetados para suportar um tipo de negócio em particular, atividade de negócio ou, por vezes, uma empresa completa. Por exemplo, um sistema de solução COTS pode ser produzido para dentistas que lidam com consultas, registros dentais, recuperação de pacientes etc. Em uma escala maior, um sistema de Enterprise Resource Planning (ERP) pode apoiar toda a fabricação, os pedidos e as atividades de gerenciamento de relacionamento com o cliente, em uma grande empresa.

Sistemas de solução COTS de domínios específicos, como sistemas para dar suporte a uma função de negócios (por exemplo, gerenciamento de documentos), fornecem funções suscetíveis de serem requeridas por uma variedade de potenciais usuários. No entanto, eles também incorporam suposições internas sobre como trabalham os usuários, e estes podem causar problemas em situações específicas. Por exemplo, um sistema para oferecer suporte à inscrição de alunos em uma universidade pode assumir que os alunos serão registrados para um diploma em uma faculdade. No entanto, se as universidades colaboram para oferecer diplomas conjuntos, então pode ser praticamente impossível representar os diplomas nesse sistema.

Os sistemas ERP, assim como aqueles produzidos pela SAP e BEA, são sistemas integrados em grande escala, projetados para oferecer suporte a práticas de negócios, como encomenda e faturamento, gerenciamento de inventário e programação de manufatura (O'LEARY, 2000). Para esses sistemas, o processo de configuração envolve a coleta de informações detalhadas sobre negócios do cliente e processos de negócios e a incorporação destes em um banco de dados de configuração, o que exige, muitas vezes, o conhecimento detalhado da configuração, notações e ferramentas, e geralmente é realizado por consultores e clientes.

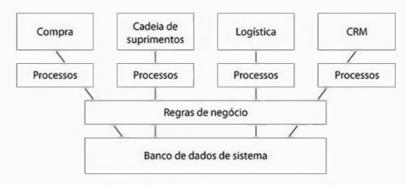
Um sistema ERP genérico inclui uma série de módulos que podem ser compostos de diferentes maneiras para a criação de um sistema para um cliente. O processo de configuração envolve a escolha de quais módulos devem ser incluídos, a configuração desses módulos individuais, a definição de processos de negócios e regras de negócios e definição da estrutura e organização do banco de dados do sistema. Na Figura 16.8 é apresentado um modelo de arquitetura geral de um sistema ERP que oferece suporte a uma variedade de funções de negócio.

As principais características dessa arquitetura são:

1. Um número de módulos para oferecer suporte a funções de negócios diferentes. Esses são módulos de alta granularidade que podem suportar todo departamento ou divisões de negócios. No exemplo da Figura 16.8, os módulos que foram selecionados para inclusão no sistema são: um módulo para apoiar a compra, um módulo para oferecer suporte a cadeia de suprimentos, um módulo de logística para apoiar a entrega de mercadorias e um módulo de gerenciamento de relacionamento com o cliente (CRM, do inglês *costumer relationship management*) para manter informações sobre os clientes.

Tabela 16.4 Sistemas de solução COTS e integrados COTS

Sistemas de solução COTS	Sistemas integrados COTS
Único produto que oferece a funcionalidade necessária para um cliente.	Vários produtos de sistemas heterogêneos são integrados para fornecer funcionalidade customizada.
Baseados em uma solução genérica e em processos padronizados.	Podem ser desenvolvidas soluções flexíveis para processos do cliente.
O foco do desenvolvimento é a configuração de sistema.	O foco do desenvolvimento é a integração do sistema.
O fornecedor do sistema é responsável pela manutenção.	O proprietário do sistema é responsável pela manutenção.
O fornecedor do sistema fornece a plataforma para o sistema.	O proprietário do sistema fornece a plataforma para o sistema.

Figura 16.8 A arquitetura de um sistema ERP

2. Um conjunto definido de processos de negócios, associados a cada módulo, que se relaciona com as atividades daquele módulo. Por exemplo, pode haver uma definição do processo de encomenda que define como os pedidos são criados e aprovados. Isso especificará as funções e atividades envolvidas em se fazer um pedido.
3. Um banco de dados comum que mantém informações sobre todas as funções de negócios relacionadas. Isso significa que não deve ser necessário replicar informações, como detalhes de clientes, em diferentes partes do negócio.
4. Um conjunto de regras de negócios que se aplica a todos os dados no banco de dados. Portanto, quando ocorre a entrada de dados de uma função, essas regras devem garantir a consistência com os dados requeridos por outras funções. Por exemplo, pode haver uma regra de negócio para que todas as declarações de despesas precisem ser aprovadas por alguém superior hierarquicamente à pessoa que tenha feito a reivindicação.

Os sistemas de ERP são usados em quase todas as grandes companhias para oferecer suporte a algumas ou todas as funções. Portanto, são uma forma amplamente usada de reúso de software. No entanto, a limitação óbvia dessa abordagem de reúso é que a funcionalidade do sistema se restringe à funcionalidade do núcleo genérico. Além disso, processos e operações de uma empresa devem ser expressos na linguagem de configuração do sistema e pode haver incompatibilidade entre os conceitos de negócio e os conceitos suportados na linguagem de configuração.

Por exemplo, em um sistema ERP vendido para uma universidade, o conceito de um cliente precisava ser definido, o que causou grandes dificuldades quanto à configuração do sistema. No entanto, as universidades têm vários tipos de clientes, como estudantes, agências financiadoras de pesquisas, instituições educacionais etc., cada um com características diferentes. Nenhum deles é realmente comparável a um cliente comercial (ou seja, uma pessoa ou empresa que compra produtos ou serviços). Quando ocorre uma grave incompatibilidade entre o modelo de negócios usado pelo sistema e o comprador do sistema, existe alta probabilidade de que o sistema ERP não atenda às reais necessidades do comprador (SCOTT, 1999).

Tanto o sistema ERP quanto os produtos de COTS de domínio específico geralmente requerem extensa configuração para se adaptar aos requisitos de cada organização em que estão instalados. Essa configuração pode envolver:

1. Seleção da funcionalidade requerida do sistema (por exemplo, ao decidir quais módulos devem ser incluídos).
2. Estabelecimento de um modelo de dados que define como os dados da organização serão estruturados no banco de dados de sistema.
3. Definição de regras de negócios que se apliquem a esses dados.
4. Definição das interações esperadas, com sistemas externos.
5. Criação de formulários de entrada e os relatórios de saída gerados pelo sistema.
6. Criação de novos processos de negócios que estejam em conformidade com o modelo de processo suportado pelo sistema.
7. Configuração de parâmetros que definam como o sistema é implantado em sua plataforma subjacente.

Uma vez que as definições de configuração são concluídas, um sistema de solução COTS está pronto para o teste. O teste é um grande problema quando os sistemas estão configurados e não programados usando-se uma linguagem convencional. Como esses sistemas são construídos a partir de uma plataforma confiável, quedas e falhas de sistema óbvias são relativamente raras; contudo, muitas vezes os problemas são sutis e relacionam-se com

as interações entre os processos operacionais e a configuração de sistema. Estas podem ser detectáveis apenas pelos usuários finais e, portanto, podem não ser descobertas durante o processo de teste de sistema. Além disso, testes de unidade automatizados, suportados por testes de *frameworks* como o JUnit, não podem ser usados. É improvável que o sistema-base dê suporte a qualquer tipo de automação de teste e pode não haver uma especificação de sistema completa para derivar testes de sistema.

16.4.2 Sistemas integrados COTS

Sistemas integrados COTS são aplicações que incluem dois ou mais produtos COTS ou, às vezes, sistemas de aplicações legados. Você pode usar essa abordagem quando não houver um sistema COTS único que atenda a todas as suas necessidades ou quando você desejar integrar um novo produto COTS com sistemas que já usa. Os produtos COTS podem interagir por meio de suas APIs (interfaces de programação de aplicação, do inglês *application programming interfaces*) ou interfaces de serviço, caso estas sejam definidas. Como alternativa, podem ser compostos da conexão entre a saída de um sistema e a entrada de outro, ou da atualização dos bancos de dados usados pelas aplicações COTS.

Para desenvolver sistemas usando produtos COTS, é necessário fazer algumas opções de projeto:

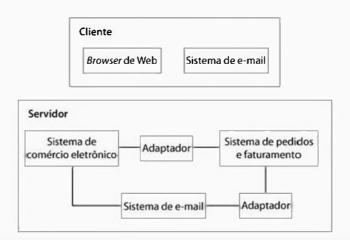
1. *Quais produtos COTS oferecem a funcionalidade mais apropriada?* Normalmente, há vários produtos COTS disponíveis, que podem ser combinados de maneiras diferentes. Se você não tiver experiência com um produto COTS, pode ser difícil decidir qual é o mais adequado.
2. *Como os dados serão trocados?* Normalmente, produtos diferentes usam formatos e estruturas de dados exclusivos. Você precisa escrever adaptadores que convertam de uma representação para outra. Esses adaptadores são sistemas em tempo de execução que operam junto com os produtos COTS.
3. *Quais características de um produto serão realmente usadas?* Um produto COTS pode ter mais funcionalidade do que você precisa e pode ser duplicado em diferentes produtos. Você precisa decidir quais recursos de quais produtos são mais adequados para seus requisitos. Se possível, você também deve negar acesso à funcionalidade não usada, pois isso pode interferir no funcionamento normal do sistema. A falha do primeiro voo do foguete Ariane 5 (NUSEIBEH, 1997) foi consequência de uma falha em um sistema de navegação inercial que foi reusado do sistema Ariane 4. No entanto, a funcionalidade que falhou não era realmente necessária no Ariane 5.

Considere o seguinte cenário como uma ilustração de uma integração COTS. Uma grande organização pretende desenvolver um sistema de aquisições que permite às pessoas fazerem pedidos a partir de suas mesas. Ao introduzir esse sistema em toda a organização, a empresa estima que economizará cinco milhões de dólares por ano. Ao centralizar a compra, o novo sistema de aquisições garante que os pedidos sejam sempre feitos a partir de fornecedores que oferecem os melhores preços e devem reduzir os custos da papelada com os pedidos. Da mesma forma com sistemas manuais, o sistema envolve que um fornecedor tenha os bens disponíveis, a criação de um pedido, a aprovação do pedido, enviar o pedido para um fornecedor, receber mercadorias e confirmar que o pagamento seja feito.

A empresa tem um sistema legado de compras usado por um escritório de aquisição. Esse processo de processamento de pedidos é integrado a um sistema de entrega e faturamento já existente. Para criar o novo sistema de aquisições, o sistema legado é integrado a uma plataforma de comércio eletrônico baseada em Web e a um sistema de correio eletrônico que trata as comunicações com os usuários. A estrutura final do sistema de aquisição, construída usando COTS, é mostrada na Figura 16.9.

Esse sistema de aquisições é baseado na arquitetura cliente-servidor, e, no cliente, o software de navegação e e-mail padrão da Web são usados. No servidor, a plataforma de comércio eletrônico precisa integrar-se com o sistema de pedidos existentes por meio de um adaptador. O sistema de comércio eletrônico tem seu próprio formato de pedidos, confirmações de entrega, e assim por diante, e estes precisam ser convertidos para o formato usado pelo sistema de pedidos. O sistema de comércio eletrônico usa o sistema de e-mail para enviar notificações aos usuários, mas o sistema de pedidos nunca foi concebido para isso. Portanto, outro adaptador precisa ser escrito para converter as notificações do sistema de pedidos em mensagens de e-mail.

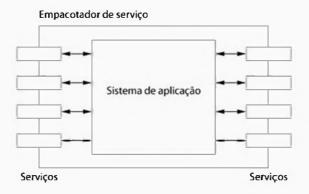
Meses, às vezes anos de esforço de implementação podem ser salvos, e o tempo para desenvolver e implantar um sistema pode ser drasticamente reduzido, usando-se uma abordagem integrada COTS. O sistema de aquisições descrito foi implementado e implantado em uma empresa muito grande em nove meses, enquanto a estimativa do tempo necessário para desenvolver o sistema em Java era de três anos.

Figura 16.9 Um sistema integrado COTS de aquisições

A integração COTS pode ser simplificada se houver uma abordagem orientada a serviços. Uma abordagem orientada a serviços significa essencialmente permitir o acesso à funcionalidade do sistema de aplicação por meio de uma interface de serviço-padrão, com um serviço para cada unidade discreta da funcionalidade. Algumas aplicações podem oferecer uma interface de serviço, mas, às vezes, essa interface precisa ser implementada pelo integrador do sistema. Essencialmente, você deve programar um empacotador que oculte a aplicação e forneça serviços externamente visíveis (Figura 16.10). Essa abordagem é particularmente valiosa para sistemas legados que precisem ser integrados com sistemas de aplicações mais recentes.

A princípio, integrar produtos COTS é o mesmo que integrar quaisquer outros componentes. Você precisa compreender as interfaces do sistema e usá-las exclusivamente para se comunicar com o software. É necessário negociar requisitos específicos por rápido desenvolvimento e reúso; e você precisa projetar uma arquitetura de sistema que permita que os sistemas COTS operem em conjunto.

No entanto, o fato de que esses produtos são eles próprios, geralmente, sistemas de grande porte e, frequentemente, são vendidos como sistemas autônomos separados, apresenta problemas adicionais. Boehm e Abts (1999) discutem quatro importantes problemas de integração de sistema COTS:

1. *Falta de controle de funcionalidade e desempenho.* Embora a interface publicada de um produto possa parecer oferecer os recursos necessários, estes podem não ser executados corretamente, ou podem executar mal. O produto pode ocultar operações que interfiram com seu uso em uma situação específica. Corrigir esses problemas pode ser uma prioridade para o integrador de produto COTS, mas pode não ser uma preocupação real para o fornecedor de produto. Os usuários podem ter de encontrar maneiras de solucionar os problemas se quiserem reusar o produto COTS.

Figura 16.10 Empacotamento de aplicação

2. *Problemas com a interoperabilidade de sistema COTS.* Algumas vezes, é difícil obter produtos COTS para trabalhar juntos porque cada produto incorpora as próprias suposições sobre como será usado. Garlan e colegas (1995), ao relatarem sua experiência em tentar integrar quatro produtos COTS, descobriram que três desses produtos foram baseados em eventos, mas cada um usava um modelo diferente de eventos. Cada sistema assumiu que tinha acesso exclusivo à fila de eventos. Como consequência, a integração foi muito difícil. O projeto exigiu cinco vezes mais esforço do que o que havia sido inicialmente calculado. O cronograma foi estendido em dois anos, em vez dos seis meses previstos. Em uma análise retrospectiva de seu trabalho, dez anos mais tarde, Garlan e colegas (2009) concluíram que não tinham sido resolvidos os problemas de integração que eles descobriram. Torchiano e Morisio (2004) concluíram que a falta de adesão a normas, em alguns produtos COTS, significava que a integração seria mais difícil do que o esperado.
3. *Não existe controle sobre a evolução de sistema.* Os fornecedores de produtos COTS tomam suas próprias decisões de evolução sobre alterações no sistema em resposta às pressões do mercado. Para os produtos de PC, em particular, são frequentemente produzidas novas versões, as quais podem não ser compatíveis com todas as versões anteriores. Novas versões podem ter funcionalidade adicional indesejada e versões anteriores podem tornar-se indisponíveis e sem suporte.
4. *Suporte dos fornecedores COTS.* O nível de suporte dado pelos fornecedores COTS varia muito. O suporte de fornecedor é particularmente importante quando surgem problemas, como os desenvolvedores não terem acesso ao código-fonte e à documentação detalhada do sistema. Embora os fornecedores possam se comprometer a fornecer suporte, as mudanças de mercado e circunstâncias econômicas podem dificultar que eles honrem esse compromisso. Por exemplo, um vendedor COTS pode decidir interromper um produto por causa de demanda limitada, ou ele pode ser adquirido por outra empresa que não pretende apoiar todos os produtos que tenham sido adquiridos.

Boehm e Abts consideram que, em muitos casos, o custo de manutenção e evolução de sistema pode ser maior para sistemas integrados COTS. Todas as dificuldades anteriormente mencionadas são problemas de ciclo de vida que não afetam apenas o desenvolvimento inicial do sistema. Quanto mais afastados das pessoas envolvidas na manutenção do sistema ficarem os desenvolvedores do sistema original, mais provavelmente surgirão dificuldades reais com os COTS integrados.

- Atualmente, a maioria dos novos sistemas de software de negócios é desenvolvida com reúso de conhecimento e códigos de sistemas já implementados.
- Existem muitas maneiras de se reusar um software, desde o reúso de classes e métodos em bibliotecas até o reúso de sistemas de aplicação completos.
- As vantagens do reúso de software são redução de custos, desenvolvimento de software mais rápido e diminuição de riscos, além de aumento da confiança no sistema. Ademais, especialistas podem ser usados mais eficazmente, concentrando sua *expertise* no projeto de componentes reusáveis.
- *Frameworks* de aplicações são coleções de objetos concretos e abstratos projetados para o reúso por meio da especialização e a adição de novos objetos. Geralmente, eles incorporam boas práticas de projeto por meio de padrões de projeto.
- As linhas de produtos de software são aplicações relacionadas, desenvolvidas a partir de uma ou mais aplicações-base. Um sistema genérico é adaptado e especializado para atender aos requisitos específicos de funcionalidade, plataforma de destino ou configuração operacional.
- O reúso de produtos COTS está preocupado com o reúso de sistemas de prateleira em grande escala. Eles fornecem uma grande funcionalidade e seu reúso pode reduzir radicalmente os custos e o tempo de desenvolvimento. Os sistemas podem ser desenvolvidos por meio da configuração de um único produto genérico COTS ou integrando dois ou mais produtos COTS.
- Os sistemas ERP (*Enterprise Resource Planning*) são exemplos de reúso de COTS em grande escala. Você cria uma instância de um sistema ERP por meio da configuração de um sistema genérico com informações sobre regras e processos de negócios do cliente.
- Possíveis problemas com o reúso baseado em COTS incluem a falta de controle sobre a funcionalidade e o desempenho, a falta de controle sobre a evolução de sistema, a necessidade de suporte dos fornecedores externos e as dificuldades em assegurar que os sistemas possam interoperar.

LEITURA COMPLEMENTAR

Reuse-based Software Engineering. Uma discussão abrangente sobre as diferentes abordagens para o reúso de software. Os autores abrangem questões técnicas de reúso e gerenciamento de processos de reúso. (MILI, H.; MILI, R.; YACOUB, S.; ADDY, E. *Reuse-based Software Engineering.* John Wiley & Sons, 2002.)

'Overlooked Aspects of COTS-Based Development'. Um artigo interessante que discute uma pesquisa de desenvolvedores usando uma abordagem baseada em COTS e os problemas que eles encontraram. (TORCHIANO, M.; MORISIO, M. *IEEE Software*, v. 21, n. 2, mar.-abr. 2004.) Disponível em: <http://dx.doi.org/10.1109/MS.2004.1270770>.

'Construction by Configuration: A New Challenge for Software Engineering'. Esse é um artigo no qual abordo os problemas e as dificuldades de construção de uma nova aplicação configurando sistemas existentes. (SOMMERVILLE, I. *Proc. XIX Conferência de engenharia de software australiano*, 2008.) Disponível em: <http://dx.doi.org/10.1109/ASWEC.2008.75>.

'Architectural Mismatch: Why Reuse Is Still So Hard'. Esse artigo faz uma releitura de um anterior, que discutiu os problemas de reúso e integração de vários sistemas COTS. Os autores concluíram que, embora haja alguns progressos, ainda existem problemas em suposições feitas pelos projetistas de sistemas individuais. (GARLAN, M. et al. *IEEE Software*, v. 26, n. 4, jul.-ago. 2009.) Disponível em: <http://dx.doi.org/10.1109/MS.2009.86>.

EXERCÍCIOS

16.1 Quais são os principais fatores técnicos e não técnicos que impedem o reúso de software? Você reúsa muitos softwares? Se não, por quê?

16.2 Sugira justificativas para a economia no custo de reúso de softwares existentes não ser proporcional ao tamanho dos componentes que são reusados.

16.3 Cite quatro casos em que você pode recomendar o não reúso de software.

16.4 Explique o que se entende por 'inversão de controle' em *frameworks* de aplicações. Explique por que essa abordagem poderia causar problemas se você integrasse dois sistemas separados que foram originalmente criados usando o mesmo *framework* de aplicação.

16.5 Usando o exemplo do sistema de estação meteorológica descrito nos capítulos 1 e 7, sugira uma arquitetura de linha de produto para uma família de aplicações relacionadas com a monitoração e a coleta de dados remotos. Você deve apresentar sua arquitetura como um modelo em camadas, mostrando os componentes que podem ser incluídos em cada nível.

16.6 A maioria dos softwares de desktop, como softwares de processamento de textos, podem ser configurados de várias maneiras. Examine o software que você usa regularmente e liste as opções de configuração para ele. Sugira as dificuldades que os usuários podem ter na configuração do software. Se você usa o Microsoft Office ou Open Office, esses são bons exemplos para usar neste exercício.

16.7 Por que muitas empresas grandes selecionam os sistemas ERP como base para seus sistemas de informações organizacionais? Quais problemas podem surgir durante a implantação de um sistema ERP, em larga escala, em uma organização?

16.8 Identifique seis possíveis riscos que podem surgir quando os sistemas são construídos usando COTS. Que medidas uma empresa pode tomar para reduzir esses riscos?

16.9 Explique por que, normalmente, são necessários adaptadores quando sistemas são construídos por meio da integração de produtos COTS. Sugira três problemas práticos que podem surgir na escrita de software de adaptadores para conectar dois produtos de aplicação COTS.

16.10 O reúso de softwares gera uma série de questões de direitos autorais e propriedade intelectual. Se um cliente paga um fornecedor de softwares para desenvolver um sistema, quem tem o direito de reusar o código desenvolvido? O fornecedor de softwares tem direito a usar esse código como base para um componente genérico? Quais mecanismos de pagamento podem ser usados para reembolsar os fornecedores de componentes reusáveis? Discuta essas e outras questões éticas associadas ao reúso de software.

REFERÊNCIAS

BAKER, T. Lessons Learned Integrating COTS into Systems. *Proc. ICCBSS 2002 (1st Int. Conf on COTS-based Software Systems)*, Orlando, Fla.: Springer, 2002, p. 21-30.

BALK, L. D.; KEDIA, A. PPT: A COTS Integration Case Study. *Proc. Int. Conf. on Software Eng.*, Limerick, Ireland: ACM Press, 2000, p. 42-49.

BAUMER, D.; GRYCZAN, G.; KNOLL, R.; LILIENTHAL, C.; RIEHLE, D.; ZULLIGHOVEN, H. Framework Development for Large Systems. *Comm. ACM*, v. 40, n. 10, 1997, p. 52-59.

BOEHM, B.; ABTS, C. COTS Integration: Plug and Pray?. *IEEE Computer*, v. 32, n. 1, 1999, p. 135-138.

BROWNSWORD, L.; MORRIS, E. The Good News about COTS. Disponível em: <http://www.sei.cmu.edu/news-at-sei/features/2003/1q03/feature-1-1q03.htm>.

CUSAMANO, M. The Software Factory: A Historical Interpretation. *IEEE Software*, v. 6, n. 2, 1989, p. 23-30.

FAYAD, M. E.; SCHMIDT, D. C. Object-oriented Application Frameworks. *Comm. ACM*, v. 40, n. 10, 1997, p. 32-38.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns*: Elements of Reusable Object-Oriented Software. Reading, Mass.: Addison-Wesley, 1995.

GARLAN, D.; ALLEN, R.; OCKERBLOOM, J. Architectural Mismatch: Why Reuse is so Hard. *IEEE Software*, v. 12, n. 6, 1995, p. 17-26.

______. Architectural Mismatch: Why Reuse is Still so Hard. *IEEE Software*, v. 26, n. 4, 2009, p. 66-69.

GRISS, M. L.; WOSSER, M. Making reuse work at Hewlett-Packard. *IEEE Software*, v. 12, n. 1, 1995, p. 105-107.

HOLDENER, A. T. *Ajax*: The Definitive Guide. Sebastopol, Calif.: O'Reilly and Associates, 2008.

JACOBSON, I.; GRISS, M.; JONSSON, P. *Software Reuse*. Reading, Mass.: Addison-Wesley, 1997.

MATSUMOTO, Y. Some Experience in Promoting Reusable Software: Presentation in Higher Abstract Levels. *IEEE. Trans. on Software Engineering*, v. SE-10, n. 5, 1984, p. 502-512.

McILROY, M. D. Mass-produced software components. *Proc. NATO Conf. on Software Eng.* Garmisch, Alemanha: Springer-Verlag, 1968.

NUSEIBEH, B. Ariane 5: Who Dunnit? *IEEE Software*, v. 14, n. 3, 1997, p. 15-16.

O'LEARY, D. E. *Enterprise Resource Planning Systems*: Systems, Life Cycle, Electronic Commerce and Risk. Cambridge, Reino Unido: Cambridge University Press, 2000.

PFARR, T.; REIS, J. E. The Integration of COTS/GOTS within NASA's HST Command and Control System. *Proc. ICCBSS 2002 (1st Int. Conf on COTS-based Software Systems)*. Orlando, Fla.: Springer, 2002, p. 209-221.

SCHMIDT, D. C. Applying design patterns and frameworks to develop object-oriented communications software. In: *Handbook of Programming Languages*, v. 1. Nova York: Macmillan Computer Publishing, 1997.

SCHMIDT, D. C.; GOKHALE, A.; NATARAJAN, B. Leveraging Application Frameworks. *ACM Queue*, v. 2, 5 jul./ago. 2004, p. 66-75.

SCOTT, J. E. The FoxMeyer Drug's Bankruptcy: Was it a Failure of ERP. *Proc. Association for Information Systems 5th Americas Conf. on Information Systems*. Milwaukee, WI, 1999.

TORCHIANO, M.; MORISIO, M. Overlooked Aspects of COTS-Based Development. *IEEE Software*, v. 21, n. 2, 2004, p. 88-93.

TRACZ, W. COTS Myths and Other Lessons Learned in Component-Based Software Development. In: HEINEMAN, G. T.; COUNCILL, W. T. (Orgs.). *Component-based Software Engineering*. Boston: Addison-Wesley, 2001, p. 99-112.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 **17** 18 19 20 21 22 23 24 25 26

Engenharia de software baseada em componentes

Objetivos

O objetivo deste capítulo é descrever uma abordagem sobre o reúso de software baseado na composição de componentes reusáveis, padronizados. Com a leitura deste capítulo, você:

- saberá que a engenharia de software baseada em componentes está preocupada com o desenvolvimento dos componentes padronizados baseados em modelos de componentes, além da composição destes em sistemas de aplicações;
- compreenderá o que se entende por um componente e um modelo de componente;
- conhecerá as principais atividades do processo CBSE para reúso e o processo CBSE com reúso;
- entenderá algumas das dificuldades e problemas que possam surgir durante o processo de composição de componentes.

Conteúdo

Como explicado no Capítulo 16, atualmente muitos novos sistemas de negócios são desenvolvidos pela configuração de sistemas disponíveis no mercado. No entanto, quando uma empresa não pode usar um sistema de prateleira, porque eles não atendem a seus requisitos, o software de que necessitam precisa ser especialmente desenvolvido. Para o desenvolvimento de softwares customizados, a engenharia de software baseada em componentes é uma forma eficaz, orientada ao reúso, de desenvolver novos sistemas corporativos.

A engenharia de software baseada em componentes (CBSE, do inglês *component-based software engineering*) surgiu na década de 1990 como uma abordagem para softwares de desenvolvimento de sistemas com base no reúso de componentes de softwares. Sua criação foi motivada pela frustração de projetistas, pois o desenvolvimento orientado a objetos não levou a um amplo reúso, como se havia sugerido. As classes de objetos foram muito detalhadas e específicas e muitas vezes precisavam ser associadas com uma aplicação em tempo de compilação. Era preciso ter conhecimento detalhado das classes para usá-las e isso geralmente significava que era necessário ter o código-fonte do componente, o que significava que vender ou distribuir objetos como componentes reusáveis individuais era praticamente impossível.

Os componentes são abstrações de nível mais alto do que objetos e são definidos por suas interfaces. Geralmente, eles são maiores que objetos individuais e todos os detalhes de implementação são escondidos de outros componentes. CBSE é o processo de definir, implementar, integrar ou compor componentes independentes, pouco acoplados em sistemas. Essa se tornou uma importante abordagem de desenvolvimento de software porque os sistemas de software estão ficando maiores e mais complexos. Os clientes exigem softwares mais confiáveis, entregues e implantados mais rapidamente.

A única maneira pela qual podemos lidar com a complexidade e entregar o melhor software, mais rapidamente, é reusar em vez de reimplementar os componentes de software.

Os fundamentos da engenharia de software baseada em componentes são:

1. Os componentes independentes que são completamente especificados por suas interfaces. Deve haver uma separação clara entre a interface de componente e sua implementação. Isso significa que a implementação de um componente pode ser substituída por outra, sem que se alterem outras partes do sistema.
2. Os padrões de componentes que facilitam a integração destes. Essas normas são incorporadas a um modelo de componentes. Eles definem, no mínimo, como interfaces de componentes devem ser especificadas e como os componentes se comunicam. Alguns modelos vão muito mais longe e definem as interfaces que devem ser implementadas por todos os componentes. Se os componentes estão em conformidade com os padrões, sua operação é independente de sua linguagem de programação. Componentes escritos em linguagens diferentes podem ser integrados ao mesmo sistema.
3. O *middleware* que fornece suporte de software para a integração de componentes. Para tornar independentes, os componentes distribuídos trabalham juntos; você precisa de suporte de *middleware* que lide com as comunicações de componentes. O *middleware* para suporte ao componente lida, com eficiência, com questões de nível inferior e permite que você se concentre nos problemas relacionados com a aplicação. Além disso, o *middleware* para suporte de componentes pode fornecer suporte para alocação de recursos, gerenciamento de transações, proteção e concorrência.
4. Um processo de desenvolvimento que é voltado para a engenharia de software baseada em componentes. Você precisa de um processo de desenvolvimento que permita que os requisitos evoluam, dependendo da funcionalidade dos componentes disponíveis. Na Seção 17.2, eu discuto os processos de desenvolvimento de CBSE.

O desenvolvimento baseado em componentes encarna as boas práticas da engenharia de software. Faz sentido projetar um sistema usando componentes, mesmo que seja necessário desenvolver, em vez de reusar esses componentes. A CBSE de base apoia-se em um dos princípios de projeto na construção de softwares compreensíveis e passíveis de manutenção:

1. Componentes são independentes, então eles não interferem na operação uns dos outros. Detalhes de implementação são ocultados. Implementação dos componentes pode ser alterada sem afetar o restante do sistema.
2. Os componentes comunicam-se por meio de interfaces bem definidas. Se essas interfaces forem mantidas, um componente poderá ser substituído por outro, que forneça funcionalidade adicional ou aumentada.
3. As infraestruturas dos componentes oferecem uma gama de serviços-padrão que podem ser usados em sistemas de aplicações, o que reduz a quantidade de códigos novos a serem desenvolvidos.

A motivação inicial para CBSE foi a necessidade de apoiar o reúso e a engenharia de software distribuída. Um componente era percebido como um elemento de um sistema de software que podia ser acessado pelo uso de um mecanismo de chamada de procedimento remoto, por outros componentes em execução em computadores separados. Cada sistema que reusou um componente precisava incorporar a própria cópia desse componente. Essa ideia de um componente estendia a noção de objetos distribuídos, conforme definido em modelos de sistemas distribuídos, como a especificação CORBA (POPE, 1997). Uma variedade de protocolos e normas foi desenvolvida para dar suporte a essa visão de um componente, como a Enterprise Java Beans (EJB) da Sun, COM e .NET da Microsoft, bem como CCM da CORBA (LAU e WANG, 2007).

Na prática, esses vários padrões têm dificultado a absorção de CBSE. Era impossível para os componentes desenvolvidos trabalharem juntos usando abordagens diferentes. Componentes que são desenvolvidos para diferentes plataformas, como .NET ou J2EE, não podem interoperar. Além disso, as normas e os protocolos propostos eram complexos e difíceis de serem compreendidos. Esse também foi um obstáculo a sua adoção.

Em resposta a esses problemas, a noção de um componente como um serviço foi desenvolvida e normas foram propostas para apoiar a engenharia de software orientada a serviços. A diferença mais significativa entre um componente como um serviço e a noção original de um componente é que os serviços são entidades autônomas externas ao programa que os usa. Quando você cria um sistema orientado a serviços, é melhor referenciar o serviço externo do que incluir uma cópia desse serviço em seu sistema.

A engenharia de software orientada a serviços, que discuto no Capítulo 19, é, portanto, um tipo de engenharia de software baseada em componentes. Ela usa uma noção mais simples de um componente do que o inicialmente proposto pela CBSE. Ela tem sido impulsionada, desde o início, por normas. Em situações nas quais o reúso baseado em um COTS é impraticável, a CBSE orientada a serviços está se tornando a abordagem dominante para o desenvolvimento de sistemas de negócios.

17.1 Componentes e modelos de componentes

Na comunidade CBSE, não existe consenso sobre um componente ser uma unidade independente de software que pode ser composta com outros componentes para criar um sistema de software. Além disso, as pessoas têm definições diferentes a respeito de um componente de software. Councill e Heineman (2001) definem um componente como:

> *Um elemento de software que está de acordo com um modelo de componente padrão e pode ser independentemente implantado e composto, sem modificações, de acordo com um padrão de composição.*

Essa definição é essencialmente baseada em padrões, assim, uma unidade de software que esteja em conformidade com esses padrões é um componente. No entanto, Szyperski (2002) não menciona padrões em sua definição de um componente, mas, em vez disso, concentra-se nas principais características dos componentes:

> *Um componente de software é uma unidade de composição com interfaces contratualmente especificadas e apenas dependências de contexto explícitas. Um componente de software pode ser implantado de forma independente e está sujeito a ser composto por parte de terceiros.*

Ambas as definições baseiam-se na noção de um componente como um elemento incluído em um sistema, em vez de um serviço que é referenciado pelo sistema. No entanto, elas também são compatíveis com a ideia de um serviço como um componente.

Szyperski também afirma que um componente não tem nenhum estado externamente observável. Isso significa que cópias de componentes são indistinguíveis. No entanto, alguns modelos de componentes, como o modelo de Enterprise Java Beans, permitem componentes com estado e, assim, estes não correspondem à definição de Szyperski. Embora componentes sem estado certamente sejam mais simples de usar, existem alguns sistemas em que os componentes com estado são mais convenientes e reduzem a complexidade do sistema.

O que as definições anteriormente mencionadas têm em comum é o fato de concordarem que os componentes são independentes e que, em um sistema, eles são a unidade fundamental de composição. Em minha opinião, uma melhor definição de um componente pode ser obtida combinando-se essas propostas. A Tabela 17.1 mostra o que eu considero que sejam as características essenciais de um componente como na CBSE.

Tabela 17.1 Características dos componentes

Característica do componente	Descrição
Padronizado	A padronização de componentes significa que um componente usado em um processo CBSE precisa obedecer a um modelo de componentes padrão. Esse modelo pode definir as interfaces de componentes, metadados de componente, documentação, composição e implantação.
Independente	Um componente deve ser independente, deve ser possível compor e implantá-lo sem precisar usar outros componentes específicos. Nessas situações, em que o componente precisa dos serviços prestados externamente, estes devem ser explicitamente definidos em uma especificação de interface 'requires'.
Passível de composição	Para um componente ser composto, todas as interações externas devem ter lugar por meio de interfaces publicamente definidas. Além disso, ele deve proporcionar acesso externo a informações sobre si próprio, como seus métodos e atributos.
Implantável	Para ser implantável, um componente dever ser autocontido. Deve ser capaz de operar como uma entidade autônoma em uma plataforma de componentes que forneça uma implementação do modelo de componentes, o que geralmente significa que o componente é binário e não tem como ser compilado antes de ser implantado. Se um componete é implantado como um serviço, ele não precisa ser implantado por um usuário de um componente. Pelo contrário, é implantado pelo prestador do serviço.
Documentado	Os componentes devem ser completamente documentados para que os potenciais usuários possam decidir se satisfazem a suas necessidades. A sintaxe e, idealmente, a semântica de todas as interfaces de componentes devem ser especificadas.

Uma maneira útil de se pensar sobre um componente é como um provedor de um ou mais serviços. Quando um sistema precisa de um serviço, ele chama um componente para fornecer esse serviço sem se preocupar sobre onde esse componente está em execução ou a linguagem de programação usada para desenvolvê-lo. Por exemplo, um componente de um sistema de biblioteca pode fornecer um serviço de pesquisa que permite aos usuários pesquisarem catálogos de diferentes bibliotecas. Um componente que converte de um formato gráfico para outro (por exemplo, TIFF para JPEG) fornece um serviço de conversão de dados etc.

A percepção de um componente como um provedor de serviços enfatiza duas características essenciais de um componente reusável:

1. O componente é uma entidade executável independente que é definida por suas interfaces. Para usá-lo, você não precisa de nenhum conhecimento de seu código-fonte. Ele pode ser referenciado como um serviço externo ou incluído diretamente em um programa.
2. Os serviços oferecidos por um componente são disponibilizados por meio de uma interface, e todas as interações se dão por meio dessa interface. A interface de componente é expressa em termos de operações parametrizadas e seu estado interno nunca é exposto.

Os componentes têm duas interfaces relacionadas, como mostrado na Figura 17.1. Essas interface refletem os serviços que o componente fornece e os serviços de que o componente necessita para funcionar corretamente:

- A interface 'provides' define os serviços prestados pelo componente. Essa interface, essencialmente, é API de componente. Ela define os métodos que podem ser chamados por um usuário do componente. Em um diagrama de componentes UML, a interface 'provides' para um componente é indicada por um círculo no final de uma linha a partir do ícone de componente.
- A interface 'requires' especifica quais serviços devem ser fornecidos por outros componentes no sistema se um componente deve funcionar corretamente. Se eles não estiverem disponíveis, o componente não funcionará. Isso não compromete a independência ou a capacidade de implantação de um componente, pois a interface 'requires' não define como esses serviços deverão ser prestados. Em UML, o símbolo de uma interface 'requires' é um semicírculo no final de uma linha a partir do ícone de componente. Observe que os ícones de interface 'provides' e 'requires' podem encaixar-se como uma bola e um soquete.

Para ilustrar essas interfaces, a Figura 17.2 mostra um modelo de um componente que foi projetado para coletar e reunir informações a partir de um vetor de sensores. Ele é executado autonomamente para coletar dados durante um período e, sob requisição, fornece dados agrupados para um componente que chama. A interface 'requires' inclui métodos para adicionar, remover, iniciar, parar e testar sensores. O método report retorna os dados do sensor que foram coletados e o método listAll fornece informações sobre os sensores conectados. Apesar de eu não ter mostrado isso aqui, esses métodos associaram parâmetros especificando os identificadores, locais de sensores, e assim por diante.

A interface 'requires' é usada para conectar os componentes aos sensores. Ela pressupõe que os sensores têm uma interface de dados, acessados por meio de sensorData, e uma interface de gerenciamento, acessada por meio de sensorManagement. Essa interface foi projetada para conectar-se a diferentes tipos de sensores, assim ela não inclui operações de sensores específicos, como Test, provideReading etc. Em vez disso, os comandos usados por um tipo específico de sensores são embutido sem uma *string*, que é um parâmetro para as operações na interface 'requires'. Os componentes adaptadores analisam essa *string* e traduzem os comandos embutidos para a interface de controle específica de cada tipo de sensor. Neste capítulo eu dicuto o uso de adaptadores, quando mostro como o componente coletor de dados está ligado a um sensor (Figura 17.10).

Uma diferença fundamental entre um componente como um serviço externo e um componente como um elemento de programa é que os serviços são entidades totalmente independentes. Eles não têm uma interface 'requires'. Diferentes programas podem usar esses serviços sem a necessidade de implementar qualquer suporte adicional exigido pelo serviço.

Figura 17.1 Interfaces de componentes

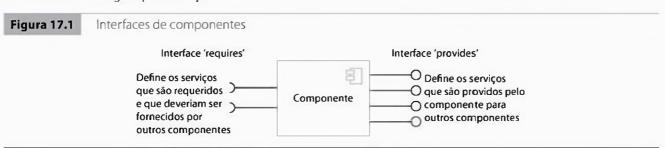
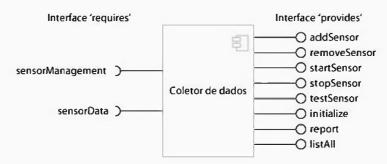
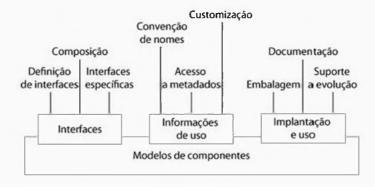



Figura 17.2 Um modelo de componente coletor de dados

17.1.1 Modelos de componentes

Um modelo de componente é uma definição de normas para a implementação, documentação e implantação de componentes. Essas normas servem para os desenvolvedores de componentes garantirem que os componentes podem interoperar. Elas também existem para os fornecedores de infraestruturas de execução de componentes que oferecem suporte à operação de componentes. Muitos modelos de componentes foram propostos, mas, atualmente, os modelos mais importantes são o modelo WebServices, o modelo Enterprise Java Beans (EJB) da Sun e o modelo .NET da Microsoft (LAU e WANG, 2007).

Os elementos básicos de um modelo ideal de componentes são discutidos por Weinreich e Sametinger (2001). Eu sumarizo esses elementos de modelo na Figura 17.3, em um diagrama que mostra que os elementos de um modelo de componente definem as interfaces de componentes, as informações que você precisa para usar o componente em um programa e como um componente deve ser implantado.

1. *Interfaces*. Os componentes são definidos pela especificação de suas interfaces. O modelo de componente especifica como as interfaces devem ser definidas e os elementos, como nomes de operação, parâmetros e exceções, que devem ser incluídos na definição de interface. O modelo também deve especificar a linguagem usada para definir as interfaces de componente. Para *web services*, é o WSDL (do inglês Web Services Description Language), que discuto no Capítulo 19; o EJB é específico de Java, então o Java é usado como a linguagem de definição de interface; no .NET, as interfaces são definidas usando-se a Linguagem Intermediária Comum (CIL, do inglês *Common Intermediate Language*). Alguns modelos de componentes exigem interfaces específicas que devem ser definidas por um componente. Esses modelos são usados para compor o componente com a infraestrutura de modelo de componente, que fornece serviços padronizados, como gerenciamento de proteção e transação.
2. *Uso*. Para que componentes sejam distribuídos e acessados remotamente, eles precisam ter um nome exclusivo ou identificador associado a eles. Isso deve ser globalmente exclusivo — por exemplo, no EJB, um nome hierárquico é gerado com a raiz baseada em um nome de domínio de Internet. Os serviços têm um único URI (*Uniform Resource Identifier*).

Figura 17.3 Elementos básicos de um modelo de componentes

Metadados de componentes são dados sobre o componente em si, como informações sobre suas interfaces e atributos. Os metadados são importantes porque permitem aos usuários do componente descobrirem quais serviços são providos e requeridos. Implementações do modelo de componente, normalmente, incluem maneiras específicas (como o uso de uma interface de reflexão em Java) para acessar esse componente metadados.

Os componentes são entidades genéricas e, quando implantados, precisam ser configurados para caber em um sistema de aplicação. Por exemplo, você poderia configurar o componente Data collector (Figura 17.1) definindo o número máximo de sensores em um vetor de sensores. O modelo de componente, portanto, pode especificar como os componentes binários podem ser customizados para um ambiente de implantação específico.

3. *Implantação*. O modelo de componente inclui uma especificação de como componentes devem ser empacotados para implantação como entidades independentes, executáveis. Como os componentes são entidades independentes, eles precisam ser empacotados com todos os softwares de suporte não fornecidos pela infraestrutura de componente ou não serão definidos em uma interface 'requires'. Informações de implantação incluem informações sobre o conteúdo de um pacote e sua organização binária.

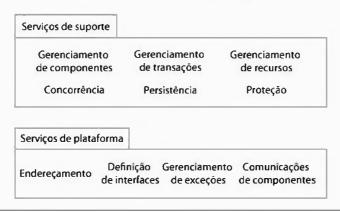
Sempre que surgirem novos requisitos, inevitavelmente os componentes terão de ser alterados ou substituídos. O modelo de componentes, portanto, pode incluir regras que gorvernam quando e como é permitida a substituição de componentes. Finalmente, o modelo de componente pode definir a documentação do componente que deve ser produzida. Isso é usado para localizar o componente e decidir se ele é apropriado.

Para componentes implementados como unidades de programa, em vez de serviços externos, o modelo de componente define os serviços a serem fornecidos pelo *middleware* que ofereça suporte à execução de componentes. Weinreich e Sametinger (2001) usam a analogia de um sistema operacional para explicar modelos de componentes. Um sistema operacional fornece um conjunto de serviços genéricos que podem ser usados pelas aplicações. Uma implementação do modelo de componente fornece serviços compartilhados comparáveis para componentes. Figura 17.4 mostra alguns dos serviços que podem ser fornecidos por uma implementação de um modelo de componente.

Os serviços fornecidos por uma implementação de modelo de componente dividem-se em duas categorias:

1. Serviços de plataforma, que permitem aos componentes se comunicarem e interagirem em um ambiente distribuído. Esses são os serviços fundamentais que devem estar disponíveis em todos os sistemas baseados em componentes.
2. Serviços de suporte, que são serviços comuns, suscetíveis de serem requeridos por muitos componentes diferentes. Por exemplo, muitos componentes requerem autenticação para garantir que o usuário dos serviços de componente seja autorizado. Faz sentido fornecer um conjunto-padrão de serviços de *middleware* para uso de todos os componentes, o que reduz os custos de desenvolvimento de componentes e as potenciais incompatibilidades de componentes podem ser evitadas.

O *middleware* implementa os serviços de componente e fornece interfaces para esses serviços. Para fazer uso dos serviços previstos por uma infraestrutura de modelo de componentes, você pode pensar em componentes para serem implantados em um 'contêiner'. Um contêiner é uma implementação dos serviços de suporte mais uma definição das interfaces que um componente deve fornecer para integrá-lo com o contêiner. Incluir um componente no contêiner significa que o componente pode acessar os serviços de suporte e o contêiner pode

Figura 17.4 Serviços de *middleware* definidos em um modelo de componente

acessar as interfaces de componente. Quando em uso, as interfaces de componentes próprios não são acessadas diretamente por outros componentes. Em vez disso, elas são acessadas por meio de uma interface de contêiner que chama código para acessar a interface de componente embutido.

Os contêiners são grandes e complexos e, quando você implanta um componente em um contêiner, você tem acesso a todos os serviços de *middleware*. No entanto, os componentes simples podem não precisar de todas as facilidades oferecidas pelo suporte de *middleware*. A abordagem adotada em *web services* para prestação de serviços comuns, portanto, é um pouco diferente. Para *web services*, padrões foram definidos para serviços comuns, como gerenciamento de transações e proteção, e esses padrões têm sido implementados como bibliotecas de programa. Caso você esteja implementando um componente de serviço, use apenas os serviços comuns de que você necessita.

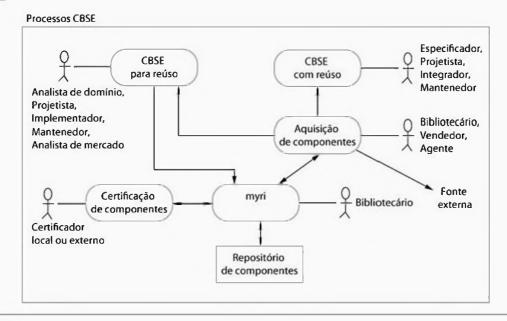
17.2 Processos CBSE

Os processos CBSE são processos de software que oferecem suporte a engenharia de software baseada em componentes. Consideram as possibilidades de reúso e as diferentes atividades do processo envolvidas no desenvolvimento e uso de componentes reusáveis. A Figura 17.5 (KOTONYA, 2003) apresenta uma visão geral dos processos CBSE. No nível mais alto, existem dois tipos de processos CBSE:

1. *Desenvolvimento para reúso.* Esse processo está interessado no desenvolvimento de componentes ou serviços que serão reusados em outras aplicações. Esse processo geralmente envolve generalizar os componentes existentes.
2. *Desenvolvimento com reúso.* Esse é o processo de desenvolvimento de novas aplicações usando componentes e serviços existentes.

Esses processos têm objetivos diferentes e, portanto, incluem atividades diferentes. No desenvolvimento por processo de reúso, o objetivo é produzir um ou mais componentes reusáveis. Você conhece os componentes com os quais trabalhará, além de ter acesso a seu código-fonte para generalizá-lo. Em desenvolvimento com reúso, você não sabe quais componentes estão disponíveis, por isso você precisa descobrir esses componentes e projetar seu sistema para fazer o uso mais eficiente deles. Você não pode ter acesso ao código-fonte do componente.

Na Figura 17.5, você pode ver que os processos básicos CBSE com e para reúso apoiam os processos que estão preocupados com a aquisição de componente, gerenciamento de componente e certificação de componente:

Figura 17.5 Processos CBSE

1. Aquisição de componente é o processo de aquisição de componentes para reúso ou desenvolvimento em um componente reusável. Pode envolver acesso a componentes desenvolvidos localmente ou serviços, ou encontrar esses componentes em uma fonte externa.
2. O gerenciamento de componente está preocupado com o gerenciamento de componentes reusáveis da empresa, garantindo que eles sejam devidamente catalogados, armazenados e disponibilizados para reúso.
3. A certificação do componente é o processo de verificação e certificação de que um componente atende a sua especificação.

Os componentes mantidos por uma organização podem ser armazenados em um repositório de componentes que inclui os componentes e as informações sobre seu uso.

17.2.1 CBSE para reúso

CBSE para reúso é o processo de desenvolver componentes reusáveis e torná-los disponíveis para reúso por meio de um sistema de gerenciamento de componentes. A visão dos primeiros defensores da CBSE (SZYPERSKI, 2002) foi a de que se desenvolveria um próspero mercado de componentes. Haveria provedores de componentes especializados e fornecedores de componentes que organizariam a venda de componentes de diferentes desenvolvedores. Os desenvolvedores de software comprariam componentes para incluir em um sistema ou pagar por serviços enquanto estes fossem usados. No entanto, essa visão não se concretizou. Relativamente, existem poucos fornecedores de componentes e a compra deles é incomum. Até o momento, o mercado de serviços também é pouco desenvolvido, apesar das previsões de significativa expansão nos próximos anos.

Por conseguinte, é mais provável que CBSE para reúso ocorra em uma organização que tenha o compromisso de usar a engenharia de software orientada a reúso. Eles pretendem explorar os ativos de software que foram desenvolvidos em diferentes partes da empresa. No entanto, esses componentes desenvolvidos internamente em geral não são reusáveis sem alterações. Muitas vezes, eles incluem recursos específicos da aplicação e interfaces que não são suscetíveis de serem necessárias em outros programas nos quais o componente seja reusado.

Para fazer componentes reusáveis você precisa se adaptar e estender os componentes específicos da aplicação para criar versões mais genéricas e, portanto, mais reusáveis. Certamente, essa adaptação tem um custo associado. Assim, você precisa decidir, em primeiro lugar, se um componente é suscetível de ser reusado, e em segundo lugar, se a economia de custos do reúso futuro justifica os custos de fazer o componente reusável.

Para responder à primeira dessas perguntas você precisa decidir se o componente implementa uma ou mais abstrações de domínio estável. As abstrações de domínio estável são elementos fundamentais de domínio da aplicação, os quais mudam lentamente. Por exemplo, em um sistema bancário, abstrações de domínio podem incluir contas, titulares de contas e demonstrações. Em um sistema de gerenciamento hospitalar, abstrações de domínio podem incluir pacientes, tratamentos e enfermeiros. Essas abstrações de domínio, às vezes, são chamadas 'objetos de negócios'. Se o componente é uma implementação de uma abstração de domínio usada com frequência ou grupo de objetos de negócios relacionados, ele provavelmente pode ser reusado.

Para responder à pergunta sobre a relação custo-benefício você deve avaliar os custos das mudanças necessárias para tornar o componente reusável. Tratam-se dos custos da documentação de componente, validação de componente e referentes a tornar o componente mais genérico. As alterações necessárias a um componente para torná-lo mais reusável incluem:

- remoção de métodos específicos de aplicação;
- alteração de nomes para torná-los mais gerais;
- adicionar métodos para fornecer uma cobertura funcional mais completa;
- tornar o tratamento de exceções consistente para todos os métodos;
- adicionar uma interface 'configuração' para permitir que o componente possa ser adaptado a diferentes situações de uso;
- integrar componentes necessários para aumentar a independência.

O problema de tratamento de exceções é particularmente difícil. Os componentes devem, eles próprios, tratar exceções, porque cada aplicação terá seus próprios requisitos para tratamento de exceção. Em vez disso, o componente deve definir quais exceções podem surgir e publicá-las como parte da interface. Por exemplo, um componente simples que implemente uma estrutura de dados de pilha deve detectar e publicar exceções de *overflow* e *underflow* de pilha. No entanto, na prática, existem dois problemas com isso:

1. Publicar todas as exceções leva a interfaces inchadas, que são mais difíceis de serem compreendidas. Isso pode afastar os potenciais usuários do componente.
2. O funcionamento do componente pode depender de tratamento de exceção local e mudar isso pode ter sérias implicações para a funcionalidade do componente.

Mili et al. (2002) discutem formas de estimar os custos de fazer um componente reusável, bem como os retornos do investimento. Os benefícios de reusar, em vez de desenvolver um componente, não são apenas os ganhos de produtividade. Também existem ganhos de qualidade, porque um componente reusado deve ser mais confiável, além dos ganhos de tempo de entrada no mercado. Esses são elementos que aumentam o retorno que acumula com a implantação do software mais rapidamente. Mili et al. (2002) apresentam várias fórmulas para calcular esses ganhos, como faz o modelo COCOMO, discutido no Capítulo 23 (BOEHM et al., 2000). No entanto, os parâmetros dessas fórmulas são difíceis de serem estimados com precisão, e as fórmulas devem ser adaptadas às circunstâncias locais, o que dificulta seu uso. Suspeito que alguns gerentes de projeto de software usem esses modelos para estimar o retorno sobre o investimento a partir da reusabilidade de componentes.

Certamente, a possibilidade de reúso de um componente depende de seu domínio de aplicação e funcionalidade. À medida que você adiciona generalidades a um componente, você aumenta sua capacidade de reúso. No entanto, isso normalmente significa que o componente tem mais operações e é mais complexo, o que torna o componente mais difícil de ser entendido e usado.

Portanto, existe um compromisso inevitável entre o reúso e a usabilidade de um componente. Para tornar um componente reusável, você deve fornecer um conjunto de interfaces genéricas com operações que atendam a todas as possibilidades de uso de um componente. Tornar o componente usável significa fornecer uma interface simples, mínima, que seja de fácil compreensão. A reusabilidade adiciona complexidade e, portanto, reduz a compreensibilidade do componente. Por conseguinte, é mais difícil decidir quando e como reusar esse componente. Ao projetar um componente reusável, você deve, portanto, encontrar um equilíbrio entre a generalidade e a compreensibilidade.

Uma potencial fonte de componentes é a existência de sistemas legados. Conforme discutido no Capítulo 9, tratam-se de sistemas que cumprem uma importante função de negócios, mas que são escritos usando-se tecnologias obsoletas de software. Por causa disso, pode ser difícil usá-los com novos sistemas. No entanto, se você converter esses sistemas antigos para componentes, sua funcionalidade pode ser reusada em novas aplicações.

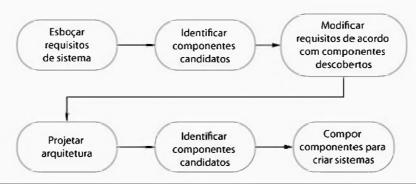
Esses sistemas legados em geral não definem claramente interfaces 'requires' e 'provides'. Para tornar esses componentes reusáveis, você deve criar um empacotador que defina as interfaces de componente. O empacotador oculta a complexidade do código subjacente e fornece uma interface para componentes externos acessarem os serviços fornecidos. Embora esse empacotador seja um software bastante complexo, o custo de seu desenvolvimento é, frequentemente, muito menor que o custo de reimplementação do sistema legado. Eu discuto essa abordagem mais detalhadamente no Capítulo 19, no qual explico como os recursos em um sistema legado podem ser acessados por meio de serviços.

Depois de ter sido desenvolvido e testado um componente ou serviço reusável, este deve ser gerenciado para reúso futuro. O gerenciamento envolve decidir como classificar o componente para que ele possa ser descoberto, de modo que o componente fique disponível tanto em um repositório como em um serviço, que informações sobre o uso do componente sejam mantidas e que o controle sobre diferentes versões do componente seja preservado. Se o componente for *open source*, você pode torná-lo disponível em um repositório público, tal como Sourceforge. Se ele se destinar para uso em uma empresa, você poderá usar um sistema de repositório interno.

Uma empresa com um programa de reúso pode realizar alguma forma de certificação de componente antes que o componente seja disponibilizado para reúso. A certificação significa que alguém diferente do desenvolvedor verifica a qualidade do componente. Eles testam o componente e certificam se ele atingiu um padrão de qualidade aceitável, antes de ser disponibilizado para reúso. No entanto, isso pode ser um processo caro e muitas empresas simplesmente deixam os testes de qualidade para os desenvolvedores de componentes.

17.2.2 CBSE com reúso

O reúso bem-sucedido de componentes requer um processo de desenvolvimento sob medida para CBSE. A CBSE com o processo de reúso precisa incluir atividades que encontrem e integrem componentes reusáveis. A estrutura desse processo foi discutida no Capítulo 2. A Figura 17.6 mostra as principais atividades dentro desse processo. Algumas delas, como a descoberta inicial de requisitos de usuário, são executadas da mesma forma

Figura 17.6 CBSE com reúso

como em outros processos de software. No entanto, as diferenças essenciais entre CBSE com reúso e processos de software para desenvolvimento de software original são:

1. Os requisitos de usuário são inicialmente desenvolvidos em alto nível, em vez de detalhes, e os *stakeholders* são incentivados a serem tão flexíveis quanto possível na definição de seus requisitos. Os requisitos muito específicos limitam o número de componentes que poderiam atender a esses requisitos. No entanto, ao contrário do desenvolvimento incremental, você precisa de um conjunto completo de requisitos para poder identificar o maior número possível de componentes para reúso.
2. Requisitos são refinados e modificados no processo de acordo com os componentes disponíveis. Se os requisitos de usuário não podem ser satisfeitos por componentes disponíveis, você deve discutir os requisitos relacionados que podem ser suportados. Os usuários podem estar dispostos a mudar de opinião se isso significar a entrega mais barata ou mais rápida do sistema.
3. Após a arquitetura de sistema ser projetada, existe uma atividade adicional de refinamento da pesquisa e projeto de componente. Alguns componentes aparentemente usáveis podem vir a ser impróprios ou a não funcionar corretamente com outros componentes escolhidos. Embora a Figura 17.6 não mostre, isso implica que podem ser necessárias mudanças nos requisitos adicionais.
4. O desenvolvimento é um processo de composição em que os componentes descobertos são integrados. Trata-se de integrar os componentes com a infraestrutura de modelo de componente e, muitas vezes, desenvolver adaptadores que conciliem as interfaces dos componentes incompatíveis. Claro que funcionalidade adicional pode ser necessária acima e por cima daquelas fornecidas pelos componentes reusáveis.

O estágio de projeto de arquitetura é particularmente importante. Jacobson et al. (1997) descobriram que definir uma arquitetura robusta é crítico para reúso com êxito. Durante a atividade de projeto de arquitetura, você pode escolher um modelo de componente e uma plataforma de implementação. No entanto, muitas empresas têm uma plataforma de desenvolvimento padrão (por exemplo, .NET), assim o modelo de componente é predeterminado. Conforme discutido no Capítulo 6, nesse estágio você também estabelece a organização de alto nível do sistema e toma decisões sobre a distribuição e o controle de sistema.

Uma atividade exclusiva para o processo CBSE é identificar componentes ou serviços candidatos ao reúso. Isso envolve uma série de subatividades, como mostrado na Figura 17.7. Inicialmente, seu foco deve ser a busca e seleção. Você precisa convencer a si mesmo de que existem componentes disponíveis para atender a seus requisitos. Certamente, você deve fazer alguma verificação inicial de que o componente seja adequado, mas testes detalhados podem não ser necessários. Em estágios posteriores, depois que a arquitetura do sistema tenha sido projetada, você deve despender mais tempo na validação de componente. É necessário estar confiante de que os componentes identificados são realmente adequados para sua aplicação. Se não, você precisa repetir os processos de busca e seleção.

Figura 17.7 O processo de identificação de componentes

O primeiro passo na identificação de componentes é olhar para os componentes ou fornecedores confiáveis disponíveis localmente. Como eu disse na seção anterior, existem, relativamente, poucos fornecedores de componentes e, portanto, é possível que você procure por componentes que foram desenvolvidos em sua própria empresa. As empresas de desenvolvimento de software podem construir sua própria base de dados de componentes reusáveis, sem os riscos inerentes ao uso de componentes de fornecedores externos. Como alternativa, você pode decidir procurar bibliotecas de código disponíveis na Web, como o Sourceforge ou Google Code, para ver se o código-fonte para o componente que você precisa está disponível. Se você estiver procurando por serviços, existem inúmeros motores de busca na Web especializados capazes de descobrir *web services* públicos.

Uma vez que o processo de busca de componentes tenha identificado possíveis componentes, você deve selecionar componentes candidatos para avaliação. Em alguns casos, essa será uma tarefa simples; os componentes da lista implementarão os requisitos de usuário e não haverá competição entre os componentes que correspondam a esses requisitos. Em outros casos, no entanto, o processo de seleção poderá ser muito mais complexo; não haverá um mapeamento claro de requisitos para componentes e você poderá achar que vários componentes precisam ser integrados para atender a um requisito específico ou a um grupo de requisitos. Por isso, terá de decidir quais composições de componentes fornecem a melhor cobertura dos requisitos.

Uma vez que você tenha selecionado componentes para uma possível inclusão em um sistema, você deve validá-los para verificar se eles se comportam como anunciado. A extensão da validação necessária depende da fonte dos componentes. Se estiver usando um componente que foi desenvolvido por uma fonte confiável e conhecida, você pode decidir que o teste de componente é desnecessário. Você testa o componente apenas quando ele é integrado com outros componentes. Por outro lado, se você estiver usando um componente de origem desconhecida, deve sempre verificar e testar esse componente antes de o incluir em seu sistema.

A validação de componente envolve o desenvolvimento de um conjunto de casos de teste para um componente (ou, possivelmente, estendendo os casos de teste fornecidos com esse componente) e o desenvolvimento de um equipamento de teste para executar testes de componentes. O grande problema com validação de componente é que a especificação de componente pode não ser suficientemente detalhada para permitir que você desenvolva um conjunto completo de testes de componentes. Geralmente, os componentes são especificados informalmente, sendo sua especificação de interface a única documentação formal . Isso pode não incluir informações suficientes para você desenvolver um conjunto completo de testes que o convenceria de que a interface do componente anunciado é o que você precisa.

Além de testar se um componente para reúso faz o que você precisa, você também deve verificar se o componente não inclui qualquer código malicioso ou funcionalidade de que você não precisa. Os desenvolvedores profissionais raramente usam componentes de fontes não confiáveis, especialmente se essas fontes não fornecem o código-fonte. Portanto, o problema de código malicioso não é comum. No entanto, componentes podem, muitas vezes, conter funcionalidade que você não precisa e você deve verificar se essa funcionalidade não interferirá com o uso do componente.

O problema com funcionalidade desnecessária é que ela pode ser ativada pelo próprio componente, o que pode retardá-lo, levá-lo a produzir resultados surpreendentes ou, em alguns casos, causar falhas graves do sistema. O Quadro 17.1 resume uma situação na qual funcionalidade desnecessária em um sistema reusado causou uma falha catastrófica de software.

Quadro 17.1 Um exempo de falha de validação com software reusado

Ao desenvolver o lançador do Ariane 5, os projetistas decidiram reusar o software de referência inercial que teve sucesso no lançador do Ariane 4. O software de referência inercial mantém a estabilidade do foguete. Eles decidiram reusar esse software sem alterações (como você faria com componentes), embora tenham incluído uma funcionalidade adicional, a qual não foi exigida no Ariane 5.

No primeiro lançamento do Ariane 5, o software de navegação inercial falhou e o foguete não pode ser controlado. Os controladores em terra instruíram o foguete a se autodestruir e sua carga foi destruída.

A causa do problema foi uma exceção não tratada quando ocorreu uma conversão de um número de ponto fixo para um número inteiro que resultou em um *overflow* numérico. Isso levou o sistema de *run-time* a desligar o sistema de referência inercial e a estabilidade do foguete não pôde ser mantida.

Essa falha nunca aconteceu no Ariane 4, porque este tinha motores menos potentes e o valor convertido não pôde ser grande o suficiente para a conversão causar *overflow*.

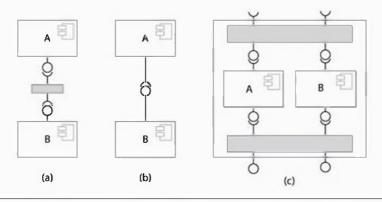
O defeito aconteceu em um código que não era necessário no Ariane 5. Os testes de validação para o reúso de software basearam-se nos requisitos do Ariane 5. Como não havia requisito para a função que falhou, não foram realizados testes. Consequentemente, o problema com o software nunca foi descoberto nos testes de simulação de lançamento.

O problema no lançador do Ariane 5 surgiu porque as suposições feitas sobre o software para o Ariane 4 eram inválidas para o Ariane 5. Esse é um problema geral com componentes reusáveis. Originalmente, eles são implementados para um ambiente de aplicações e incorporam pressupostos sobre esse ambiente. Esses pressupostos raramente são documentados, então, quando o componente é reusado, é impossível derivar testes para verificar se os pressupostos ainda são válidos. Se você vai reusar um componente em um ambiente diferente, é possível que não descubra os pressupostos de ambientes embutidos até que use o componente em um sistema operacional.

17.3 Composição de componentes

A composição de componentes é o processo de integração de componentes uns com os outros e com o componente especialmente escrito '*glue code*' para criar um sistema ou outro componente. Existem várias maneiras pelas quais você pode compor componentes, como mostrado na Figura 17.8. Da esquerda para a direita, esses diagramas ilustram a composição sequencial, a composição hierárquica e a composição aditiva. Na discussão a seguir, eu assumo que você está compondo dois componentes (A e B) para criar um novo componente:

1. Na Figura 17.8, a composição sequencial está apresentada no item (a). Você cria um novo componente a partir de dois componentes existentes, por chamar os componentes existentes em sequência. Você pode pensar a composição como uma composição de 'interfaces provides'. Ou seja, os serviços oferecidos pelo componente A são chamados e os resultados retornados por A são usados na chamada para os serviços oferecidos pelo componente B. Os componentes não chamam uns aos outros na composição sequencial. Algum '*glue code*' extra é necessário para chamar os serviços de componente na ordem certa e garantir que os resultados entregues por componente sejam compatíveis com as entradas esperadas pelo componente B. A interface 'provides' da composição depende da funcionalidade combinada de A e B, mas, normalmente, essa não será uma composição de suas interfaces 'provides'. Esse tipo de composição pode ser usado com componentes que são elementos de programa ou componentes que são serviços.
2. Na Figura 17.8, a composição hierárquica está apresentada no item (b). Esse tipo de composição ocorre quando um componente chama diretamente os serviços prestados por outro componente. O componente chamado fornece os serviços necessários para o componente chamador. Portanto, a interface 'provides' do componente chamado deve ser compatível com a interface 'requires' do componente chamador. O componente A chama diretamente o componente B e, se suas interfaces corresponderem, pode não haver necessidade de código adicional. No entanto, se houver uma incompatibilidade entre a interface 'requires' de A e a interface 'provides' de B, então algum código de conversão pode ser necessário. Como serviços não têm uma interface 'requires', esse modo de composição não é usado quando componentes são implementados como *web services*.
3. A composição aditiva corresponde à situação (c) na Figura 17.8. Ela ocorre quando dois ou mais componentes são colocados juntos (adicionados) para se criar um novo componente, que combina suas funcionalidades. A interface 'provides' e a interface 'requires' do novo componente é uma combinação das interfaces correspondentes nos componentes A e B. Os componentes são chamados separadamente por meio da interface externa do componente composto. A e B não são dependentes e não chamam uns aos outros. Esse tipo de composição pode ser usado com componentes que são unidades de programas ou componentes que são serviços.

Figura 17.8 Tipos de composição de componentes

Você pode usar todas as formas de composição de componentes durante a criação de um sistema. Em todos os casos, talvez seja necessário escrever um '*glue code*' para ligar os componentes. Por exemplo, para uma composição sequencial, a saída do componente A normalmente se torna a entrada do componente B. Você precisa de instruções intermediárias que chamem o componente A, coletem o resultado e, em seguida, chamem o componente B, com esse resultado como um parâmetro. Quando um componente chama outro, talvez você precise introduzir um componente intermediário que garanta que a interface 'provides' e a interface 'requires' sejam compatíveis.

Quando você escreve novos componentes especialmente para a composição, você deve criar as interfaces desses componentes de maneira que sejam compatíveis com outros componentes do sistema. Portanto, você pode compor esses componentes facilmente em uma única unidade. No entanto, quando os componentes são desenvolvidos para reúso de forma independente, você frequentemente é confrontado com as incompatibilidades de interfaces. Isso significa que as interfaces dos componentes que você deseja compor não são as mesmas. Podem ocorrer três tipos de incompatibilidades:

1. *Incompatibilidade de parâmetro*. As operações de cada lado da interface têm o mesmo nome, mas com tipos de parâmetro ou número de parâmetros diferentes.
2. *Incompatibilidade de operação*. Os nomes das operações nas interfaces 'provides' e 'requires' são diferentes.
3. *Incompletude de operação*. O funcionamento da interface 'provides' de um componente é um subconjunto da interface 'requires' de outro componente ou vice-versa.

Em todos os casos, você pode resolver o problema da incompatibilidade escrevendo um adaptador que reconcilia as interfaces dos dois componentes reusados. Um componente adaptador converte uma interface para outra. A forma exata do adaptador depende do tipo de composição. Às vezes, como no exemplo a seguir, o adaptador leva um resultado de um componente e converte em um formulário, que pode ser usado como entrada para outro. Em outros casos, o adaptador pode ser chamado pelo componente A como um *proxy* para o componente B. Essa situação ocorre se A pretende chamar B, mas os detalhes da interface 'requires' de A não coincidem com os detalhes da interface 'provides' de B. O adaptador reconcilia essas diferenças, convertendo os parâmetros de entrada de A para os parâmetros de entrada requeridos por B. Ele, em seguida, chama B para fornecer os serviços requeridos por A.

Para ilustrar os adaptadores, considere os dois componentes mostrados na Figura 17.9, cujas interfaces são incompatíveis. Eles podem ser parte de um sistema usado pelos serviços de emergência. Quando o operador de emergência recebe uma chamada, o número do telefone é a entrada para o componente `addressFinder` localizar o endereço. Em seguida, usando o componente mapper, o operador imprime um mapa para ser enviado para o veículo, despachado para a situação de emergência. Na verdade, os componentes teriam interfaces mais complexas do que as mostradas aqui, mas a versão simplificada ilustra o conceito de um adaptador.

O primeiro componente, `addressFinder`, localiza o endereço que corresponde a um número de telefone. Ele também pode retornar o dono da propriedade associado ao número de telefone e tipo de propriedade. O componente `mapper` leva um código postal (nos Estados Unidos, um código postal padrão com os quatro dígitos adicionais identificando o local da propriedade) e exibe ou imprime um mapa das ruas da área em torno desse código em uma escala especificada.

Em princípio, esses componentes são compostos porque o local da propriedade inclui o código postal, ou CEP. No entanto, você precisa escrever um componente adaptador chamado `postCodeStripper` que leva os dados

Figura 17.9 Componentes com interfaces incompatíveis

do local do addressFinder e remove o código postal. Então, esse código postal é usado como entrada para o mapper, e o mapa das ruas é exibido em uma escala de 1:10.000. O código a seguir, que é um exemplo de composição sequencial, ilustra a sequência de chamadas que é requerida para implementar isto:

```
address = addressFinder.location (phonenumber) ;

postCode = postCodeStripper.getPostCode (address) ;

mapper.displayMap(postCode, 10 000) ;
```

Outro caso em que um componente adaptador pode ser usado na composição hierárquica é quando um componente pretende fazer uso de outro, mas há uma incompatibilidade entre a interface 'provides' e a interface 'requires' dos componentes na composição. Na Figura 17.10, eu ilustro o uso de um adaptador; no caso, um adaptador é usado para conectar um componente coletor de dados e um sensor. Eles poderiam ser usados na implementação de um sistema de estação meteorológica no deserto, conforme discutido no Capítulo 7.

Os componentes coletor de dados e sensor são compostos usando um adaptador que reconcilia a interface 'requires' do componente coletor de dados com a interface 'provides' do componente sensor. O componente coletor de dados foi projetado com uma interface genérica 'requires' que suporta a coleta de dados e o gerenciamento de sensor. Para cada uma dessas operações, o parâmetro é uma *string* de texto que representa os comandos específicos do sensor. Por exemplo, para emitir um comando de coleta, você diria `sensorData`('collect'). Como mostrado na Figura 17.10, o próprio sensor tem operações distintas, como iniciar, parar e coletar dados.

O adaptador analisa a *string* de entrada, identifica o comando (por exemplo, coletar) e, em seguida, chama o `Sensor.getdata` para coletar o valor do sensor. Em seguida, ele retorna o resultado (como uma *string* de caracteres) para o componente coletor de dados. Esse estilo de interface significa que o coletor de dados pode interagir com diferentes tipos de sensores. Um adaptador separado, que converte os comandos de sensor a partir do `Data-collector` para a interface de sensor real, é implementado para cada tipo de sensor.

Essa discussão, sobre composição de componentes, assume que você pode dizer, a partir da documentação do componente, se a interface é compatível ou não. Naturalmente, a definição de interface inclui os tipos de parâmetros e o nome da operação, para que você possa fazer alguma avaliação de sua compatibilidade. No entanto, você depende da documentação de componente para decidir se as interfaces são semanticamente compatíveis.

Para ilustrar esse problema, considere a composição mostrada na Figura 17.11. Esses componentes são usados para implementar um sistema que transfere imagens de uma câmera digital e as armazena em uma biblioteca de fotografias. O usuário do sistema pode fornecer informações adicionais para descrever e catalogar a fotografia. Para evitar a confusão, eu não mostro nesta obra todos os métodos de interface. Em vez disso, mostro simplesmente os métodos necessários para ilustrar o problema da documentação de componentes. Os métodos na interface da Biblioteca de Fotografias são:

```
public void addItem (Identifier pid ; Photograph p; CatalogEntry photodesc) ;

public Photograph retrieve (Identifier pid) ;

public CatalogEntry catEntry (Identifier pid) ;
```

Figura 17.10 Um adaptador conectando o coletor de dados e um sensor

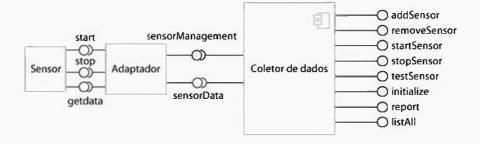
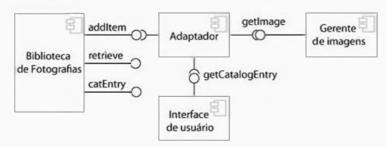



Figura 17.11 Composição de uma biblioteca de fotografias

Suponha que a documentação para o método addItem na Biblioteca da Fotografias seja:

Esse método adiciona uma fotografia para a biblioteca e associa o identificador de fotografia e o descritor de catálogo com a fotografia.

Essa descrição parece explicar o que o componente faz, mas considere as seguintes perguntas:

- O que acontece se o identificador de fotografia já estiver associado com uma fotografia na biblioteca?
- O descritor de fotografia, assim como a fotografia, é associado com a entrada do catálogo? Ou seja, se você excluir a fotografia, também exclui as informações do catálogo?

Na descrição informal não existem informações suficientes de addItem para responder a essas perguntas. Naturalmente, é possível adicionar mais informações para a descrição da linguagem natural do método, mas em geral a melhor maneira de resolver as ambiguidades é usar uma linguagem formal para descrever a interface. A especificação mostrada no Quadro 17.2 é parte da descrição da interface da `Photo Library` que adiciona informações para a descrição informal.

A especificação no Quadro 17.2 usa pré e pós-condições definidas em uma notação baseada na linguagem de restrição de objeto (OCL, do inglês *object constraint language*), que faz parte da UML (WARMER e KLEPPE, 2003). A OCL é projetada para descrever restrições nos modelos de objeto da UML; permite expressar predicados que devem ser verdadeiros sempre, que devem ser verdadeiros antes que um método seja executado e que devem ser verdadeiros após um método ser executado. Essas são invariantes, pré-condições e pós-condições. Para acessar o valor de uma variável antes de uma operação, você adiciona @pré após seu nome. Portanto, usando a idade como um exemplo:

```
idade = idade@pre + 1
```

Essa declaração significa que o valor da idade, depois de uma operação, é um a mais do que era antes da operação.

As abordagens baseadas em OCL estão cada vez mais sendo usadas para adicionar informações semânticas aos modelos da UML e as descrições OCL podem ser usadas para dirigir geradores de códigos na engenharia orientada a modelos. A abordagem geral foi obtida a partir do Projeto por Contrato de Meyer (MEYER, 1992), em que as

Quadro 17.2 A descrição OCL da interface da `Photo Library`

```
- O contexto dos nomes-chave do componente ao qual as condições se aplicam ao contexto AddItem
- As pré-condições especificam o que deve ser verdadeiro antes da execução de addItem
pre:    PhotoLibrary.libSize( ) > 0
        PhotoLibrary.retrieve (pid) = null
- As pós-condições especificam o que é verdadeiro após a execução
post:   libSize ( ) = libSize( )@pre + 1
        PhotoLibrary.retrieve(pid), p =
        PhotoLibrary.catEntry(pid) = photodesc
        context delete
pre:    PhotoLibrary.retrieve (pid) < >null ;
post:   PhotoLibrary.retrieve(pid) = null
        PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
        PhotoLibrary.libSize( ) = libSize( )@pre-1
```

interfaces e as obrigações dos objetos que se comunicam são formalmente especificadas e impostas pelo sistema de *run-time*. Meyer sugere que o uso de Projeto por Contrato é essencial se quisermos desenvolver componentes confiáveis (MEYER, 2003).

O Quadro 17.2 inclui uma especificação para os métodos addItem e delete na `Photo Library`. O método que está sendo especificado é indicado pelo contexto de palavra-chave e as pré e pós-condições, pelas palavras-chave *pre* e *post*. As pré-condições para addItem afirmam que:

1. Não deve haver, na biblioteca, uma fotografia com o mesmo identificador que a fotografia que será inserida.
2. A biblioteca deve existir — assuma que ao criar uma biblioteca você deve adicionar a ela, pelo menos, um único item, de maneira que o tamanho de uma biblioteca seja sempre maior que zero.
3. As pós-condições para addItem afirmam que:

   O tamanho da biblioteca aumentou em um (tão somente uma única entrada foi feita).

   Se você buscar usando o mesmo identificador, você recebe de volta a fotografia que adicionou.

   Se você pesquisar o catálogo usando esse identificador, você recebe de volta a entrada de catálogo que você fez.

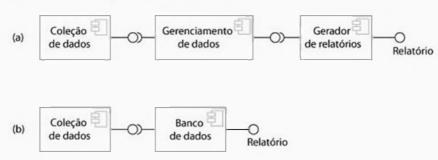
A especificação de delete fornece mais informações. A pré-condição afirma que, para um item ser deletado, ele deve estar na biblioteca e que, após ser deletada, a foto já não pode ser recuperada e o tamanho da biblioteca será reduzido em um. No entanto, deletar a foto não deleta a entrada de catálogo — você ainda poderá recuperá-la depois de a foto ter sido deletada. A razão para isso é que você pode desejar manter, no catálogo, informações sobre por que uma foto foi excluída, seu novo local, e assim por diante.

Ao criar um sistema de composição de componentes, você pode achar que não existem conflitos potenciais entre os requisitos funcionais e não funcionais, a necessidade de entregar um sistema tão rapidamente quanto possível e a necessidade de criar um sistema que possa evoluir de acordo com a mudança de requisitos. As decisões que você pode ter de levar em consideração são:

1. Qual composição de componentes é mais eficaz na entrega dos requisitos funcionais para o sistema?
2. Qual composição dos componentes tornará mais fácil adaptar os componentes compostos quando seus requisitos mudarem?
3. Quais serão as propriedades emergentes do sistema composto? Essas são propriedades como desempenho e confiabilidade. Você só pode avaliá-las depois que o sistema completo tenha sido implementado.

Infelizmente, existem muitas situações em que as soluções para os problemas de composição podem entrar em conflito. Por exemplo, considere uma situação como a ilustrada na Figura 17.12, em que um sistema pode ser criado por meio de duas composições alternativas. O sistema é uma coleção de dados e sistema de relatórios em que os dados são coletados de fontes diferentes e armazenados em um banco de dados e, em seguida, são produzidos relatórios diferentes desses dados.

Aqui, existe um potencial conflito entre o desempenho e a capacidade de adaptação. A composição (a) é mais adaptável, mas a composição (b) talvez seja mais rápida e mais confiável. As vantagens da composição (a) é que os relatórios e o gerenciamento de dados são separados e, assim, há mais flexibilidade para mudanças futuras. O sistema de gerenciamento de dados pode ser substituído e, se forem requisitados relatórios que o componente de relatório atual não pode produzir, esse componente também pode ser substituído sem a necessidade de alteração do componente de gerenciamento de dados.

Figura 17.12 Componentes da coleção de dados e geração de relatório

Na composição (b), um componente de banco de dados com recursos internos de geração de relatórios (por exemplo, o Microsoft Access) é usado. A principal vantagem da composição (b) é que existem menos componentes; assim, ela será implementada mais rapidamente, pois não há *overheads* de comunicação de componente. Além disso, as regras de integridade de dados que se aplicam ao banco de dados também serão aplicáveis aos relatórios. Esses relatórios não serão capazes de combinar dados de forma incorreta. Na composição (a) não existem essas restrições de modo que pudessem ocorrer erros nos relatórios.

Em geral, um princípio da boa composição é o princípio de separação de interesses. Ou seja, você deve tentar projetar seu sistema de forma que cada componente tenha seu papel claramente definido e que, idealmente, tais papéis não se sobreponham. No entanto, pode ser mais barato comprar um componente multifuncional, em vez de dois ou três componentes separados. Além disso, quando vários componentes são usados pode haver sanções na confiança ou no desempenho.

PONTOS IMPORTANTES

- A engenharia de software baseada em componentes é uma abordagem baseada no reúso para definição, implementação e composição de componentes independentes, fracamente acoplados em sistemas.
- Um componente é uma unidade de software cuja funcionalidade e dependências são completamente definidas por um conjunto de interfaces públicas. Os componentes podem ser compostos com outros componentes sem o conhecimento de sua implementação e podem ser implantados como uma unidade executável.
- Os componentes podem ser implementados como unidades de programa que são incluídas em um sistema ou como serviços externos que são referenciados a partir de dentro de um sistema.
- Um modelo de componente define um conjunto de padrões para componentes, incluindo normas de interface, normas de uso e normas de implantação. A implementação do modelo de componente fornece um conjunto de serviços comuns que podem ser usados por todos os componentes.
- Durante o processo CBSE, você precisa intercalar os processos de engenharia de requisitos e projeto de sistemas. Você precisa negociar os requisitos desejáveis pelos serviços que estão disponíveis a partir dos componentes reusáveis existentes.
- A composição do componente é o processo de 'conexão' dos componentes para a criação de um sistema. Os tipos de composições incluem a composição sequencial, a composição hierárquica e a composição aditiva.
- Ao compor componentes reusáveis que não foram escritos para sua aplicação, você talvez precise escrever adaptadores ou '*glue code*' para reconciliar as diferentes interfaces de componentes.
- Ao escolher as composições, você deve considerar a funcionalidade requerida para o sistema, os requisitos não funcionais e a facilidade com que um componente pode ser substituído quando o sistema é alterado.

LEITURA COMPLEMENTAR

Component-based Software Engineering: Putting the Pieces Together. Esse livro é uma coleção de artigos de vários autores sobre os diferentes aspectos da CBSE. Como todas as coleções, ela é um pouco variada, mas traz uma melhor discussão sobre as questões gerais de engenharia de software com componentes do que o livro do Szyperski. (HEINEMAN, G. T.; COUNCILL, W. T. *Component-based Software Engineering*: Putting the Pieces Together. Addison-Wesley, 2001.)

Component Software: Beyond Object-Oriented Programming, 2nd.ed. Essa edição atualizada do primeiro livro sobre CBSE abrange questões técnicas e não técnicas sobre CBSE. Ela traz mais detalhes das tecnologias específicas que o livro de Heineman e Councill, além de incluir uma discussão aprofundada das questões de mercado. (SZYPERSKI, C. *Component Software*: Beyond Object-Oriented Programming. 2. ed. Addison-Wesley, 2002.)

'Specification, Implementation and Deployment of Components'. Uma boa introdução aos fundamentos da CBSE. A mesma edição da *CACM* inclui artigos sobre componentes e desenvolvimento baseado em componentes. (CRNKOVIC, I.; HNICH, B.; JONSSON, T.; KIZILTAN, Z. *Comm. ACM*, v. 45, n. 10, out. 2002.) Disponível em: <http://dx.doi.org/10.1145/570907.570928>.

'Software Component Models'. Essa é uma discussão abrangente sobre os modelos de componentes comerciais e de pesquisa que classifica esses modelos e explica as diferenças entre eles. (LAU, K.-K.; WANG, Z. *IEEE Transactions on Software Engineering*, v. 33, n. 10, out. 2007.) Disponível em: <http://dx.doi.org/10.1109/TSE.2007.70726>.

EXERCÍCIOS

17.1 Por que é importante que todas as interações de componente sejam definidas por meio das interfaces 'requires' e 'provides'?

17.2 O princípio da independência de componentes significa que deveria ser possível substituir um componente por outro, implementado de maneira completamente diferente. Usando um exemplo, explique como tal substituição de componentes poderia ter consequências indesejadas e levar à falha de sistema.

17.3 Quais são as diferenças fundamentais entre componentes como elementos de programa e componentes como serviços?

17.4 Por que é importante que os componentes se baseiem em um modelo de componente padrão?

17.5 Usando um exemplo de um componente que implementa um tipo abstrato de dado, como uma pilha ou uma lista, mostre por que, geralmente, é necessário estender e adaptar componentes para o reúso.

17.6 Explique por que é difícil validar um componente reusável sem o código-fonte do componente. De que maneiras uma especificação formal de componentes simplificaria os problemas de validação?

17.7 Projete a interface 'provides' e a interface 'requires' de um componente reusável que possa ser usado para representar um paciente em MHC-PMS.

17.8 Usando exemplos, ilustre os diferentes tipos de adaptadores necessários para oferecer suporte à composição sequencial, à composição hierárquica e à composição aditiva.

17.9 Projete as interfaces de componentes que podem ser usadas em um sistema para uma sala de controle de emergência. Você deve projetar interfaces para um componente de registro de chamadas que registre as chamadas realizadas e um componente de descoberta do veículo que, dado um código postal (CEP) e um incidente, encontre o veículo adequado mais próximo a ser despachado para o incidente.

17.10 Foi sugerido que uma autoridade independente de certificação deveria ser estabelecida. Fornecedores submeteriam seus componentes a essa autoridade, que validaria se o componente era confiável. Quais seriam as vantagens e as desvantagens de tal autoridade de certificação?

REFERÊNCIAS

BOEHM, B. W.; ABTS, C.; BROWN, A. W.; CHULANI, S.; CLARK, B. K.; HOROWITZ, E. et al. *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ.: Prentice Hall, 2000.

COUNCILL, W. T.; HEINEMAN, G. T. Definition of a Software Component and its Elements. In: HEINEMAN, G. T.; COUNCILL, W. T. (Orgs). *Component-based Software Engineering*. Boston: Addison-Wesley, 2001, p. 5-20.

JACOBSON, I.; GRISS, M.; JONSSON, P. Software Reuse. Reading, Mass.: Addison-Wesley, 1997.

KOTONYA, G. The CBSE Process: Issues and Future Visions. *Proc. 2a CBSEnet workshop*, Budapeste, Hungria, 2003.

LAU, K.-K.; WANG, Z. Software Component Models. *IEEE Trans. on Software Eng.*, v. 33, n. 10, 2007, p. 709-724.

MEYER, B. Design by Contract. *IEEE Computer*, v. 25, n. 10, 1992, p. 40-51.

MEYER, B. The Grand Challenge of Trusted Components. *ICSE 25*: *Int. Conf. on Software Eng*, Portland, Oregon: IEEE Press, 2003.

MILI, H.; MILI, A.; YACOUB, S.; ADDY, E. *Reuse-based Software Engineering*. Nova York: John Wiley & Sons, 2002.

POPE, A. *The CORBA Reference Guide*: Understanding the Common Object Request Broker Architecture. Harlow, Reino Unido: Addison-Wesley, 1997.

SZYPERSKI, C. *Component Software*: Beyond Object-oriented Programming, 2a ed. Harlow, Reino Unido: Addison-Wesley, 2002.

WARMER, J.; KLEPPE, A. *The Object Constraint Language*: Getting your models ready for MDA. Boston: Addison-Wesley, 2003.

WEINREICH, R.; SAMETINGER, J. Component Models and Component Services: Concepts and Principles. In: HEINEMAN, G. T.; COUNCILL, W. T. (Orgs). *Component-Based Software Engineering*. Boston: Addison-Wesley, 2001, p. 33-48.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 **18** 19 20 21 22 23 24 25 26

Engenharia de software distribuído

Objetivos

O objetivo deste capítulo é introduzir engenharia de sistemas distribuídos e arquiteturas de sistemas distribuídos. Com a leitura deste capítulo, você:

- conhecerá as questões fundamentais que devem ser consideradas ao se projetarem e implementarem sistemas de software distribuídos;
- entenderá o modelo de computação cliente-servidor e a arquitetura em camadas de sistemas cliente-servidor;
- terá sido apresentado aos padrões mais comumente usados em arquiteturas de sistemas distribuídos e conhecerá os tipos de sistema para os quais cada arquitetura é mais aplicável;
- compreenderá o conceito de software como um serviço, fornecendo acesso baseado na Web para sistemas de aplicações implantados remotamente.

Conteúdo

Praticamente todos os grandes sistemas computacionais agora são sistemas distribuídos. Um sistema distribuído é um sistema que envolve vários computadores, em contraste com sistemas centralizados, em que todos os componentes do sistema executam em um único computador. Tanenbaum e Van Steen (2007) definem um sistema distribuído como:

> ... *uma coleção de computadores independentes que aparece para o usuário como um único sistema coerente.*

Obviamente, a engenharia de sistemas distribuídos tem muito em comum com a engenharia de qualquer outro software. No entanto, existem questões específicas que precisam ser consideradas ao projetar esse tipo de sistema. Estas surgem porque os componentes do sistema podem ser executados em computadores gerenciados de forma independente e porque eles se comunicam por meio de uma rede.

Coulouris et al. (2005) identificam as seguintes vantagens da utilização de uma abordagem distribuída de desenvolvimento de sistemas:

1. *Compartilhamento de recursos.* Um sistema distribuído permite o compartilhamento de recursos de hardware e software — tais como discos, impressoras, arquivos e compiladores — que estão associados em computadores em uma rede.
2. *Abertura.* Geralmente, os sistemas distribuídos são sistemas abertos, o que significa que são projetados para protocolos-padrão que permitem que os equipamentos e softwares de diferentes fornecedores sejam combinados.

3. *Concorrência*. Em um sistema distribuído, vários processos podem operar simultaneamente em computadores separados, na rede. Esses processos podem (mas não precisam) comunicar-se uns com os outros durante seu funcionamento normal.
4. *Escalabilidade*. Em princípio, pelo menos, os sistemas distribuídos são escaláveis, assim, os recursos do sistema podem ser aumentados pela adição de novos recursos para fazer face às novas exigências do sistema. Na prática, a rede que liga os computadores individuais ao sistema pode limitar a escalabilidade deste.
5. *Tolerância a defeitos*. A disponibilidade de vários computadores e o potencial para replicar as informações significa que os sistemas distribuídos podem ser tolerantes com algumas falhas de hardware e software (veja o Capítulo 13). Na maioria dos sistemas distribuídos, um serviço de má qualidade pode ser fornecido quando ocorrem falhas; a perda total do serviço só ocorre quando há uma falha de rede.

Para sistemas organizacionais de grande porte, essas vantagens significam que os sistemas distribuídos substituíram, em grande medida, os sistemas legados de *mainframe* que foram desenvolvidos na década de 1990. No entanto, existem muitos sistemas computacionais de aplicação pessoal (por exemplo, sistemas de edição de foto) que não são distribuídos e que executam em um único computador. A maioria dos sistemas embutidos também é de sistemas de processador único.

Os sistemas distribuídos são, inerentemente, mais complexos que os sistemas centralizados, o que os torna mais difíceis para projetar, implementar e testar. É mais difícil compreender as propriedades emergentes de sistemas distribuídos por causa da complexidade das interações entre os componentes do sistema e sua infraestrutura. Por exemplo, em vez de o desempenho do sistema depender da velocidade de execução de um processador, ele depende da largura da banda de rede, da carga de rede e da velocidade de todos os computadores que fazem parte do sistema. Mover os recursos de uma parte do sistema para outra pode afetar significativamente o desempenho do sistema.

Além disso, como todos os usuários da internet sabem, sistemas distribuidos têm respostas imprevisíveis. O tempo de resposta depende da carga geral sobre o sistema, sua arquitetura e a carga de rede. Como todos esses podem mudar em um curto período, o tempo necessário para responder a uma solicitação do usuário varia drasticamente de uma solicitação para outra.

O mais importante desenvolvimento que tem afetado os sistemas de software distribuído, nos últimos anos, é a abordagem orientada a serviços. Grande parte deste capítulo se concentra nas questões gerais de sistemas distribuídos, mas discuto a noção de aplicações implantadas como serviços na Seção 18.4. Esse material complementa o material do Capítulo 19, que se concentra em serviços como componentes de uma arquitetura orientada a serviços e questões mais gerais de engenharia de software orientada a serviços.

18.1 Questões sobre sistemas distribuídos

Conforme discutido na introdução deste capítulo, os sistemas distribuídos são mais complexos do que os executados em um único processador. Esta complexidade surge porque é praticamente impossível ter um modelo de controle *top-down* desses sistemas. Muitas vezes, os nós do sistema que fornecem a funcionalidade são sistemas independentes com nenhuma autoridade sobre eles. A rede conectando esses nós é um sistema gerenciado separadamente. Esse é um sistema complexo em si mesmo, que não pode ser controlado pelos proprietários dos sistemas que usam a rede. Portanto, essa é uma imprevisibilidade inerente à operação de sistemas distribuídos, a qual deve ser considerada pelo projetista do sistema.

Algumas das questões mais importantes de projeto, que devem ser consideradas nos sistemas distribuídos, são:

1. *Transparência*. Em que medida o sistema distribuído deve aparecer para o usuário como um único sistema? Quando é útil aos usuários entender que o sistema é distribuído?
2. *Abertura*. Um sistema deveria ser projetado usando protocolos-padrão que ofereçam suporte à interoperabilidade ou devem ser usados protocolos mais especializados que restrinjam a liberdade do projetista?
3. *Escalabilidade*. Como o sistema pode ser construído para que seja escalável? Ou seja, como todo o sistema poderia ser projetado para que sua capacidade possa ser aumentada em resposta às crescentes exigências feitas ao sistema?
4. *Proteção*. Como podem ser definidas e implementadas as políticas de proteção que se aplicam a um conjunto de sistemas gerenciados independentemente?
5. *Qualidade de serviço*. Como a qualidade do serviço que é entregue aos usuários do sistema deve ser especificada e como o sistema deve ser implementado para oferecer uma qualidade aceitável e serviço para todos os usuários?

6. *Gerenciamento de falhas*. Como as falhas do sistema podem ser detectadas, contidas (para que elas tenham efeitos mínimos em outros componentes do sistema) e reparadas?

Em um mundo ideal, o fato de um sistema ser distribuído seria transparente para os usuários. Isso significa que os usuários veriam o sistema como um único sistema cujo comportamento não é afetado pela maneira como o sistema é distribuído. Na prática, isso é impossível de se alcançar. O controle central de um sistema distribuído é impossível, e, como resultado, os computadores individuais em um sistema podem ter comportamentos diferentes em momentos diferentes. Além disso, como sempre leva um tempo determinado para os sinais viajarem através de uma rede, os atrasos na rede são inevitáveis. O comprimento desses atrasos depende da localização dos recursos no sistema, da qualidade da conexão da rede do usuário e da carga de rede.

A abordagem de projeto para atingir a transparência depende de se criarem abstrações dos recursos em um sistema distribuído de modo que a realização física desses recursos possa ser alterada sem a necessidade de alterações no sistema de aplicação. O *middleware* (discutido na Seção 18.1.2) é usado para mapear os recursos lógicos referenciados por um programa para os recursos físicos reais e para gerenciar as interações entre esses recursos.

Na prática, é impossível fazer um sistema completamente transparente e, geralmente, os usuários estão conscientes de que estão lidando com um sistema distribuído. Assim, você pode decidir que é melhor expor a distribuição aos usuários; eles, por sua vez, podem ser preparados para algumas das consequências da distribuição, como atrasos na rede, falhas de nó remoto etc.

Os sistemas distribuídos abertos são sistemas construídos de acordo com normas geralmente aceitas. Isso significa que os componentes de qualquer fornecedor podem ser integrados ao sistema e podem interoperar com outros componentes do sistema. Atualmente, no nível de rede, com sistemas se conformando aos protocolos de Internet, a abertura é tida como confirmada, mas no nível de componente, a abertura ainda não é universal. A abertura implica que os componentes de sistema possam ser desenvolvidos independentemente em qualquer linguagem de programação e, se estas estiverem em conformidade com as normas, funcionarão com outros componentes.

O padrão CORBA (POPE, 1997) desenvolvido na década de 1990, tinha esse objetivo, mas nunca alcançou uma massa crítica de usuários. Em vez disso, muitas empresas optaram por desenvolver sistemas usando padrões proprietários para componentes de empresas, como a Sun e a Microsoft. Estes forneciam melhores implementações e suporte de software, além de melhor suporte de longo prazo para os protocolos industriais.

Os padrões de *web services* (discutidos no Capítulo 19) para arquiteturas orientadas a serviços foram desenvolvidos para serem padrões abertos. No entanto, existe uma significativa resistência a eles por causa de sua ineficiência. Alguns desenvolvedores de sistemas baseados em serviços optaram pelos chamados protocolos RESTful porque estes têm um *overhead* inerentemente mais baixo do que os protocolos de *web services*.

A escalabilidade de um sistema reflete sua capacidade de oferecer um serviço de alta qualidade, uma vez que aumenta a demanda de sistema. Neuman (1994) identifica três dimensões da escalabilidade:

1. *Tamanho*. Deve ser possível adicionar mais recursos a um sistema para lidar com um número crescente de usuários.
2. *Distribuição*. Deve ser possível dispersar geograficamente os componentes de um sistema sem comprometer seu desempenho.
3. *Capacidade de gerenciamento*. É possível gerenciar um sistema à medida que ele aumenta de tamanho, mesmo que partes dele estejam localizadas em organizações independentes.

Em termos de tamanho, existe uma distinção entre escalamento para cima e escalamento para fora. Escalamento para cima significa a substituição de recursos no sistema por recursos mais poderosos. Por exemplo, você pode aumentar a memória em um servidor de 16 GB para 64 GB. Escalamento para fora significa adicionar recursos ao sistema (por exemplo, um servidor Web extra para trabalhar ao lado de um servidor existente). Frequentemente, o escalamento para fora é mais efetivo do que o escalamento para cima, mas, geralmente, significa que o sistema precisa ser projetado, de maneira que o processamento concorrente seja possível.

Na Parte 2 deste livro eu discuti as questões gerais de proteção e questões de engenharia de proteção. No entanto, quando um sistema é distribuído, o número de maneiras pelas quais o sistema pode ser atacado é significativamente maior, em comparação com sistemas centralizados. Se uma parte do sistema é atacada com êxito, o invasor pode ser capaz de usar isso como uma 'porta dos fundos' para outras partes do sistema.

Os tipos de ataques dos quais um sistema distribuído deve se defender são:

1. *Intercepção*, em que as comunicações entre as partes do sistema são interceptadas por um invasor de tal modo que haja uma perda de confidencialidade.
2. *Interrupção*, em que os serviços de sistema são atacados e não podem ser entregues conforme o esperado. Ataques de negação de serviços envolvem bombardear um nó com solicitações de serviço ilegítimas, para que ele não consiga lidar com solicitações válidas.
3. *Modificação*, em que os dados ou serviços no sistema são alterados por um invasor.
4. *Fabricação*, em que um invasor gera informações que não deveriam existir e, em seguida, usa-as para obter alguns privilégios. Por exemplo, um invasor pode gerar uma entrada de senha falsa e usá-la para obter acesso a um sistema.

A grande dificuldade em sistemas distribuídos é estabelecer uma política de proteção que possa ser fielmente aplicada a todos os componentes de um sistema. Conforme discutido no Capítulo 11, uma política de proteção define o nível de proteção a ser alcançado por um sistema. Mecanismos de proteção, como criptografia e autenticação, são usados para reforçar as políticas de proteção. As dificuldades em um sistema distribuído surgem porque diferentes organizações podem possuir partes do sistema. Essas organizações podem ter mecanismos de proteção e políticas de proteção incompatíveis entre si. Compromissos de proteção podem ser necessários para permitir que os sistemas trabalhem juntos.

A qualidade de serviço (QoS, do inglês *quality of service*) oferecida por um sistema distribuído reflete sua capacidade de entregar seus serviços de maneira confiável e com um tempo de resposta e taxa de transferência aceitáveis para seus usuários. Idealmente, os requisitos de QoS devem ser especificados com antecedência e o sistema, criado e configurado para entregar essa QoS. Infelizmente, isso nem sempre é possível, por duas razões:

1. Pode não ser efetivo projetar e configurar o sistema para oferecer uma alta QoS no momento de carga de pico. Isso poderia envolver disponibilizar recursos que não são usados na maior parte do tempo. Um dos principais argumentos para a 'computação em nuvem' é que ela aborda parcialmente esse problema. Usando uma nuvem, é fácil adicionar recursos conforme a demanda aumenta.
2. Os parâmetros de QoS podem ser mutuamente contraditórios. Por exemplo, maior confiabilidade pode significar taxas reduzidas de transferência, uma vez que as verificações de procedimentos sejam introduzidas para garantir que todas as entradas do sistema sejam válidas.

A QoS é particularmente crítica quando o sistema lida com dados críticos de tempo, como fluxos de som ou vídeo. Nessas circunstâncias, se a QoS cai abaixo de um valor-limite, em seguida, o som ou vídeo podem tornar-se tão degradados que se torna impossível compreendê-los. Os sistemas que lidam com som e vídeo devem incluir componentes de negociação e gerenciamento de QoS. Eles devem avaliar os requisitos de QoS comparados aos recursos disponíveis e, se forem insuficientes, negociar por mais recursos ou por um objetivo de QoS reduzido.

Em um sistema distribuído, é inevitável que ocorram falhas, assim, o sistema precisa ser projetado para ser resistente a essas falhas. A falha é tão onipresente que uma definição irreverente de um sistema distribuído sugerido por Leslie Lamport, um proeminente pesquisador de sistemas distribuídos, é:

> *Você sabe que você tem um sistema distribuído quando a parada de um sistema do que você nunca ouviu impede você de realizar qualquer trabalho.*

O gerenciamento de falhas envolve a aplicação de técnicas de tolerância a defeitos discutidas no Capítulo 13. Portanto, os sistemas distribuídos devem incluir mecanismos para descobrir se um componente do sistema falhou, devem continuar a oferecer tantos serviços quanto possível mesmo que falhe e, tanto quanto possível, devem recuperar-se automaticamente de falhas.

18.1.1 Modelos de interação

Existem dois tipos fundamentais de interação que podem ocorrer entre os computadores em um sistema de computação distribuído: interação procedural e interação baseada em mensagens. A interação procedural envolve um computador que chama um serviço conhecido oferecido por algum outro computador e (normalmente) esperando que esse serviço seja fornecido. A interação baseada em mensagens envolve o computador 'que envia' que define as informações sobre o que é requerido em uma mensagem, que são, então, enviadas para outro computador. Geralmente, as mensagens transmitem mais informações em uma única interação do que uma chamada de procedimento para outra máquina.

Para ilustrar a diferença entre a interação procedural e a interação baseada em mensagens, considere uma situação na qual você está pedindo uma refeição em um restaurante. Quando você tem uma conversa com o garçom, você está envolvido em uma série de interações síncronas, procedurais que definem seu pedido. Você faz um pedido, o garçom reconhece o pedido; você faz outra solicitação, a qual é reconhecida, e assim por diante. Isso é comparável aos componentes interagindo em um sistema de software em que um componente chama métodos de outros componentes. O garçom anota seu pedido junto com os pedidos de seus acompanhantes. Ele passa esse pedido para a cozinha, que inclui detalhes de tudo o que tenha sido ordenado, para a cozinha preparar a refeição. Essencialmente, o garçom está passando uma mensagem para o pessoal da cozinha definindo a refeição que deve ser preparada. Isso é a interação baseada em mensagens.

A Figura 18.1 ilustra isso. Ela mostra o processo síncrono de pedido como uma série de chamadas, e o Quadro 18.1 mostra uma mensagem XML hipotética que define um pedido feito por uma mesa de três pessoas. A diferença entre essas formas de intercâmbio de informações é clara. O garçom pega o pedido como uma série de interações, com cada interação definindo parte do pedido. No entanto, o garçom tem uma única interação com a cozinha, onde a mensagem define o pedido completo.

Normalmente, em um sistema distribuído, a comunicação procedural é implementada usando-se chamadas de procedimento remoto (RPCs, do inglês *remote procedure calls*). Nas RPCs, um componente chama outro componente, como se fosse um método ou procedimento local. O *middleware* no sistema intercepta essa chamada

Figura 18.1 Interação procedural entre um cliente e um garçom

Quadro 18.1 Interação baseada em mensagens entre um garçom e o pessoal da cozinha

```
<entrada>
        <nome do prato = "sopa" type = "tomate" />
        <nome do prato = "sopa" type = "peixe" />
        <nome do prato = "salada de pombo" />
</entrada>
<curso principal>
        <nome do prato = "bife" type = "lombo" cozinhar = "médio" />
        <nome do prato = "bife" type = "filé" cozinhar = "mal passado" />
        <nome do prato = "robalo">
</principal>
<acompanhamento>
        <nome do prato = "batatas fritas" porções = "2" />
        <nome do prato = "salada" porções = "1" />
</acompanhamento>
```

e transmite-a para um componente remoto. Este realiza o processamento necessário e, via *middleware*, retorna o resultado para o componente chamado. Em Java, as chamadas de método remoto (RMI, do inglês *remote method invocations*) são comparáveis às RPCs, embora não idênticas. O *framework* de RMI trata a chamada de métodos remotos em um programa em Java.

As RPCs exigem um 'stub' para o procedimento chamado ser acessível no computador que está iniciando a chamada. O stub é chamado e converte os parâmetros de procedimento em uma representação-padrão de transmissão para o procedimento remoto. Em seguida, por meio do *middleware*, envia a solicitação para execução do procedimento remoto. O procedimento remoto usa funções de biblioteca para converter os parâmetros para o formato exigido, efetua o processamento e, em seguida, comunica os resultados via 'stub' que representa o chamador.

Normalmente, a interação baseada em mensagens envolve um componente que cria uma mensagem que detalha os serviços necessários de outro componente. Através do *middleware* de sistema, ela é enviada para o componente que recebe a solicitação. O receptor analisa a mensagem, realiza os processamentos e cria uma mensagem para o componente que enviou a solicitação com os resultados desejados. Em seguida, ela é passada para o *middleware* para a transmissão para o componente que enviou a solicitação.

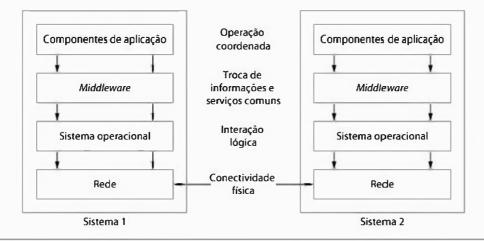
Um problema com a abordagem RPC para a interação é que o chamador e o chamado precisam estar disponíveis no momento da comunicação e devem saber como se referir um ao outro. Em essência, uma RPC tem os mesmos requisitos que uma chamada de método ou procedimento local. Por outro lado, em uma abordagem baseada em mensagens, a indisponibilidade pode ser tolerada, pois a mensagem permanece em uma fila até que o receptor esteja disponível. Além disso, não é necessário que o transmissor e o receptor da mensagem estejam conscientes um do outro. Eles se comunicam com o *middleware*, que é responsável por garantir que as mensagens sejam passadas para o sistema apropriado.

18.1.2 *Middleware*

Os componentes em um sistema distribuído podem ser implementados em diferentes linguagens de programação e podem ser executados em diferentes tipos de processador. Os modelos de dados, representação de informações, protocolos para comunicação podem ser todos diferentes. Um sistema distribuído, portanto, requer um software que possa gerenciar essas diversas partes e assegurar que elas podem se comunicar e trocar dados.

O termo '*middleware*' é usado para se referir a esse software — ele fica no meio, entre os componentes distribuídos do sistema. Isso é ilustrado na Figura 18.2, que mostra que o *middleware* é uma camada entre o sistema operacional e programas de aplicação. Normalmente, o *middleware* é implementado como um conjunto de bibliotecas que é instalado em cada computador distribuído, além de um sistema de *run-time* para gerenciar a comunicação.

Bernstein (1996) descreve os tipos de *middleware* que estão disponíveis para oferecer suporte para a computação distribuída. O *middleware* é um software de uso geral geralmente comprado no mercado e que não é escrito especialmente por desenvolvedores de aplicações. Exemplos de *middleware* incluem o software para gerenciamento de comunicações com bancos de dados, gerenciadores de transações, conversores de dados e controladores de comunicação.

Figura 18.2 O *middleware* em um sistema distribuído

Em um sistema distribuído, o *middleware* costuma fornecer dois tipos distintos de suporte:

1. Suporte a interações, em que o *middleware* coordena as interações entre diferentes componentes do sistema. O *middleware* fornece transparência da localização, assim não é necessário que os componentes saibam os locais físicos dos outros componentes. Ele também pode suportar a conversão de parâmetros se diferentes linguagens de programação forem usadas para implementar componentes, detecção de eventos e comunicação etc.
2. A prestação de serviços comuns, em que o *middleware* fornece implementações reusáveis de serviços que podem ser exigidas por vários componentes do sistema distribuído. Usando esses serviços comuns, os componentes podem, facilmente, interoperar e prestar serviços de usuário de maneira consistente.

Na Seção 18.1.1, dei exemplos do suporte a interações que o *middleware* pode fornecer. Você usa o *middleware* para suporte a chamadas de procedimento remoto e de método remoto, troca de mensagens etc.

Serviços comuns são os serviços que podem ser exigidos por componentes diferentes, independentemente da funcionalidade deles. Conforme discutido no Capítulo 17, eles podem incluir serviços de proteção (autenticação e autorização), serviços de notificação e identificação, serviços de gerenciamento de transações etc. Você pode pensar nesses serviços comuns como sendo fornecidos por um contêiner de *middleware*. Em seguida, pode implantar seu componente nesse contêiner e ele pode acessar e usar esses serviços comuns.

18.2 Computação cliente-servidor

Sistemas distribuídos que são acessados pela Internet normalmente são organizados como sistemas cliente-servidor. Em um sistema cliente-servidor, o usuário interage com um programa em execução em seu computador local (por exemplo, um *browser* de Web ou uma aplicação baseados em telefone). Este interage com outro programa em execução em um computador remoto (por exemplo, um servidor Web). O computador remoto fornece serviços, como o acesso a páginas Web, que estão disponíveis para clientes externos. Esse modelo cliente-servidor, conforme discutido no Capítulo 6, é um modelo de arquiteura geral de uma aplicação. Não está restrito a aplicações distribuídas em várias máquinas. Você também pode usá-lo como um modelo lógico de interação no qual o cliente e o servidor executam no mesmo computador.

Em uma arquitetura cliente-servidor, uma aplicação é modelada como um conjunto de serviços que são fornecidos por servidores. Os clientes podem acessar esses serviços e apresentar os resultados para os usuários finais (ORFALI e HARKEY, 1998). Os clientes precisam estar cientes dos servidores que estão disponíveis, mas não devem saber da existência de outros clientes. Clientes e servidores são processos separados, conforme mostra a Figura 18.3, a qual ilustra uma situação em que existem quatro servidores (s1–s4), que fornecem serviços diferentes. Cada serviço tem um conjunto de clientes associados que acessam esses serviços.

A Figura 18.3 mostra processos entre cliente e servidor, em vez de processadores. É normal que vários processos clientes executem em um único processador. Por exemplo, em seu PC, você pode executar um cliente de correio que transfere mensagens de um servidor de correio remoto. Você também pode executar um *browser* de Web que interage com um servidor Web remoto e um cliente de impressão que envia documentos para uma impressora remota. A Figura 18.4 ilustra a situação em que os 12 clientes lógicos mostrados na Figura 18.3 estão em execução em seis computadores. Os quatro processos servidores são mapeados em dois computadores físicos de servidor.

Figura 18.3 Interação cliente-servidor

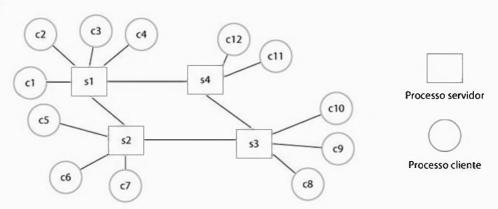
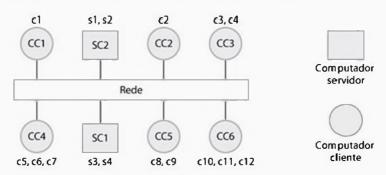



Figura 18.4 Mapeamento de clientes e servidores para computadores em rede

Vários processos servidores diferentes podem ser executados no mesmo processador, mas, muitas vezes, os servidores são implementados como sistemas multiprocessadores em que uma instância separada do processo servidor é executada em cada máquina. O software de balanceamento de carga distribui pedidos de serviço de clientes para servidores diferentes para que cada servidor realize a mesma quantidade de trabalho. Isso permite que um maior volume de transações com clientes seja manipulado, sem degradar a resposta aos clientes individuais.

Os sistemas cliente-servidor dependem de haver uma separação clara entre a apresentação de informações e as computações que criam e processam essas informações. Consequentemente, você deve projetar a arquitetura dos sitemas cliente-servidor distribuídos para que eles sejam estruturados em várias camadas lógicas, com interfaces claras entre essas camadas. Isso permite que cada camada seja distribuída para um computador diferente. A Figura 18.5 ilustra esse modelo, mostrando uma aplicação estruturada em quatro camadas:

- uma camada de apresentação que diz respeito à apresentação de informações para o usuário e gerenciamento de toda a interação com o usuário;
- uma camada de gerenciamento de dados que gerencia os dados que são passados de e para o cliente. Essa camada pode implementar verificações sobre os dados, gerar páginas Web etc.;
- uma camada de processamento de aplicação que está preocupada com a implementação da lógica da aplicação e, assim, fornece a funcionalidade necessária para os usuários finais;
- uma camada de banco de dados que armazena os dados e fornece serviços de gerenciamento de transações etc.

A seção a seguir explica como diferentes arquiteturas cliente-servidor distribuem essas camadas lógicas de maneiras diferentes. O modelo cliente-servidor também serve como base para o conceito de software como serviço (SaaS, do inglês *software as a service*), uma forma cada vez mais importante de implantação e acesso do software através da Internet. Discuto isso na Seção 18.4.

Figura 18.5 Modelo de arquitetura em camadas para aplicações cliente-servidor

18.3 Padrões de arquitetura para sistemas distribuídos

Como expliquei no início deste capítulo, os projetistas de sistemas distribuídos precisam organizar seus projetos de sistema para encontrar um equilíbrio entre desempenho, confiança, proteção e capacidade de gerenciamento do sistema. Não existe um modelo universal de organização de sistema distribuído que seja apropriado para todas as circunstâncias, de modo que surgiram vários estilos de arquitetura. Ao projetar uma aplicação distribuída, você deve escolher um estilo de arquitetura que ofereça suporte aos requisitos não funcionais críticos de seu sistema.

Nesta seção, eu discuto cinco estilos de arquitetura:

1. *Arquitetura de mestre-escravo*, é usada em sistemas de tempo real em que tempos de resposta precisos de interação são requeridos.
2. *Arquitetura cliente-servidor de duas camadas*, é usada para sistemas cliente-servidor simples e em situações nas quais é importante centralizar o sistema por razões de proteção. Nestes casos, a comunicação entre o cliente e o servidor costuma ser criptografada.
3. *Arquitetura cliente-servidor multicamadas*, é usada quando existe um alto volume de transações a serem processadas pelo servidor.
4. *Arquitetura distribuída de componentes*, é usada quando recursos de diferentes sistemas e bancos de dados precisam ser combinados ou é usada como um modelo de implementação para sistemas cliente-servidor em várias camadas.
5. *Arquitetura ponto-a-ponto*, é usada quando os clientes trocam informações localmente armazenadas e o papel do servidor é introduzir clientes uns aos outros. Ela também pode ser usada quando um grande número de computações independentes pode ser feito.

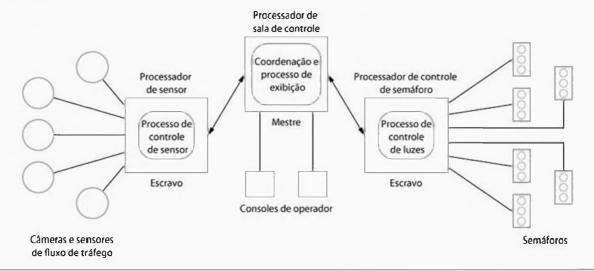
18.3.1 Arquiteturas mestre-escravos

Arquiteturas mestre-escravos para sistemas distribuídos são comumente usadas em sistemas de tempo real em que pode haver processadores separados associados à aquisição de dados do ambiente do sistema, processamento de dados, de gerenciamento de atuadores e computação. Como discutido no Capítulo 20, os atuadores são dispositivos controlados pelo sistema de software que agem para alterar o ambiente do sistema. Por exemplo, um atuador pode controlar uma válvula e alterar seu estado de 'aberta' para 'fechada'. O processo 'mestre' é geralmente responsável pelo processamento, coordenação e comunicações e controla os processos 'escravo'. Processos 'escravo' são dedicados a ações específicas, como a aquisição de dados de um vetor de sensores.

A Figura 18.6 ilustra esse modelo de arquitetura como um modelo de um sistema de controle de tráfego em uma cidade, em que três processos lógicos são executados em processadores separados. O processo mestre é o processo da sala de controle, que se comunica com processos escravos separados e que são responsáveis pela coleta de dados de tráfego e gerência de funcionamento de semáforos.

Um conjunto de sensores distribuídos coleta informações sobre o fluxo de tráfego. O processo de controle de sensores varre-os periodicamente para capturar as informações do fluxo de tráfego e confere essa informação para processamento adicional. O processador de sensor é varrido periodicamente para obtenção de informações por processo mestre que se preocupa com a exibição do *status* de tráfego para os operadores, processamento de sequências de luzes de semáforos e aceitação de comandos de operador para modificar essas sequências. O sistema de sala de controle envia comandos para um processo de controle de semáforo e converte-os em sinais para controlar o hardware das luzes de semáforos. O sistema de sala de controle mestre é organizado como um sistema cliente-servidor, com os processos clientes executando em consoles de operador.

Você pode usar esse modelo de mestre-escravo de um sistema distribuído em situações em que seja possível ou necessário prever o processamento distribuído, bem como em casos nos quais o processamento pode ser facilmente localizado para processadores escravos. Essa situação é comum em sistemas de tempo real, quando é importante cumprir os *deadlines* (prazos) de processamento. Processadores escravos podem ser usados para operações computacionalmente intensivas, como processamento de sinais e o gerenciamento de equipamentos controlados pelo sistema.

Figura 18.6 Um sistema de gerenciamento de tráfego com uma arquitetura mestre-escravo

18.3.2 Arquitetura cliente-servidor de duas camadas

Na Seção 18.2, abordei a forma geral dos sistemas cliente-servidor em que parte do sistema de aplicação é executada no computador do usuário (o cliente) e parte é executada em um computador remoto (o servidor). Apresentei também um modelo de aplicação em camadas (Figura 18.5), em que as diferentes camadas do sistema podem ser executadas em computadores diferentes.

Uma arquitetura cliente-servidor de duas camadas é a forma mais simples da arquitetura cliente-servidor. O sistema é implementado como um único servidor lógico e, também, um número indefinido de clientes que usam esse servidor. A Figura 18.7 mostra duas formas desse modelo de arquitetura:

1. *Modelo cliente-magro*, em que a camada de apresentação é implementada no cliente e todas as outras camadas (gerenciamento de dados, processamento de aplicação e banco de dados) são implementadas em um servidor. O software de cliente pode ser um programa especialmente escrito no cliente para tratar a apresentação. No entanto, é frequente um *browser* de Web no computador cliente ser usado para apresentação dos dados.
2. *Modelo cliente-gordo*, em que parte ou todo o processamento de aplicação é executado no cliente e as funções de banco de dados e gerenciamento são implementadas no servidor.

A vantagem do modelo cliente-magro é a simplicidade em gerenciar os clientes. Isso é um grande problema se houver um grande número de clientes, pois pode ser difícil e caro instalar em novo software em todos eles. Se um *browser* de Web for usado como o cliente, não é necessário instalar qualquer software.

Figura 18.7 Modelos de arquitetura cliente-magro e cliente-gordo

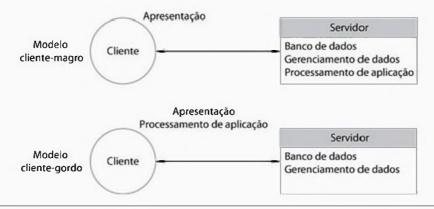
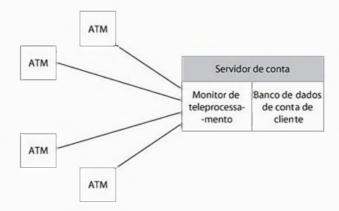



Figura 18.8 Uma arquitetura cliente-gordo para um sistema de ATM

A desvantagem da abordagem cliente-magro, porém, é poder colocar uma carga pesada de processamento no servidor e na rede. O servidor é responsável por toda a computação e isso pode levar à geração de tráfego significativo de rede entre o cliente e o servidor. A implementação de um sistema usando esse modelo, portanto, pode exigir investimentos adicionais na capacidade de rede e de servidor. No entanto, *browsers* podem efetuar algum processamento local executando *scripts* (por exemplo, Javascript) na página Web que é acessada pelo *browser*.

O modelo cliente-gordo faz uso do poder de processamento disponível no computador executando o software cliente e distribui alguns ou todo o processamento de aplicação e a apresentação para o cliente. O servidor é essencialmente um servidor de transação que gerencia todas as transações do banco de dados. O gerenciamento de dados é simples e não é necessário haver interação entre o cliente e o sistema de processamento de aplicação. Certamente, o problema com o modelo cliente-gordo é requerer gerenciamento de sistema adicional para implantar e manter o software no computador cliente.

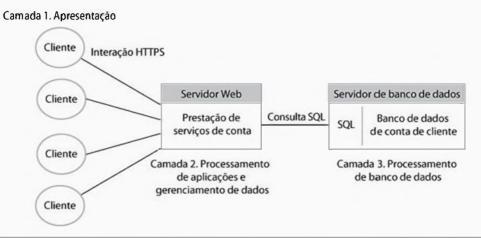
Um exemplo de uma situação em que a arquitetura cliente-gordo é usada é um sistema de banco ATM, que oferece dinheiro e outros serviços bancários para os usuários. ATM é o computador cliente, e o servidor é, normalmente, um *mainframe* executando o banco de dados de conta de cliente. Um computador *mainframe* é uma máquina poderosa que é projetada para o processamento de transações. Assim, ele pode lidar com o grande volume de transações geradas pelas ATMs e outros sistemas de caixa e bancos on-line. O software na máquina de caixa realiza vários processamentos relacionados ao cliente associado com uma transação.

A Figura 18.8 mostra uma versão simplificada da organização do sistema de uma ATM. Observe que as ATMs não estão ligadas diretamente com o banco de dados do cliente, mas sim a um monitor de teleprocessamento (TP). Um monitor de teleprocessamento é um sistema de *middleware* que organiza as comunicações com os clientes remotos e serializa as transações de clientes para o processamento no banco de dados. Isso garante que as transações sejam independentes e não interfiram entre si. Usar transações seriais significa que o sistema pode se recuperar de defeitos sem corromper os dados do sistema.

Considerando que um modelo cliente-gordo distribui o processamento mais eficazmente do que um modelo cliente-magro, o gerenciamento de sistema é mais complexo. A funcionalidade de aplicação é espalhada por muitos computadores. Quando o software de aplicação precisa ser alterado, isso envolve a reinstalação em cada computador cliente, o que pode ter um grande custo se houver centenas de clientes no sistema. O sistema pode ser projetado para oferecer suporte a atualizações de software remoto e pode ser necessário desligar todos os serviços de sistema até que o software de cliente seja substituído.

18.3.3 Arquiteturas cliente-servidor multicamadas

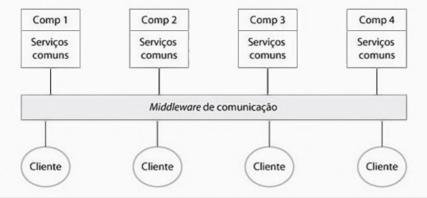
O problema fundamental com uma abordagem cliente-servidor de duas camadas é que as camadas lógicas de sistema — apresentação, processamento de aplicação, gerenciamento de dados e banco de dados — devem ser mapeadas para dois sistemas de computador: o cliente e o servidor. Pode haver problemas com escalabilidade e desempenho se o modelo cliente-magro for escolhido, ou problemas de gerenciamento de sistema se o modelo cliente-gordo for usado. Para evitar esses problemas, uma arquitetura de 'cliente-servidor multicamadas'

Figura 18.9 Arquitetura de três camadas para um sistema de *Internet banking*

pode ser usada. Nessa arquitetura, as diferentes camadas do sistema, apresentação, gerenciamento de dados, processamento de aplicação e banco de dados, são processos separados que podem ser executados em diferentes processadores.

Um sistema de *Internet banking* (Figura 18.9) é um exemplo da arquitetura cliente-servidor multicamadas, em que existem três camadas no sistema. O banco de dados de clientes do banco (geralmente, hospedado em um computador *mainframe*, como discutido anteriormente) fornece serviços de banco de dados. Um servidor Web fornece serviços de gerenciamento de dados, como a geração de página Web e alguns serviços de aplicação. Os serviços de aplicação, como recursos para transferir dinheiro, gerar extratos, pagar contas, e assim por diante, são implementados no servidor Web como *scripts*, os quais são executados pelo cliente. O computador do usuário com um *browser* de Internet é o cliente. Esse sistema é escalável, pois é relativamente fácil adicionar servidores (escalabilidade para cima), assim como o número de clientes.

Nesse caso, o uso de arquitetura de três camadas permite a transferência de informações entre o servidor Web e o servidor de banco de dados para ser otimizado. As comunicações entre esses sistemas podem usar protocolos de troca de dados rápidos e de baixo nível. Um *middleware* eficiente oferece suporte em consultas de banco de dados em SQL (Structured Query Language), sendo usado para tratar informações de recuperação do banco de dados.

O modelo cliente-servidor de três camadas pode ser estendido para uma variante em multicamadas, na qual os servidores adicionais são adicionados ao sistema. Esse processo envolve o uso de um servidor Web para gerenciamento de dados e de servidores separados para processamento de aplicação e serviços de banco de dados. Sistemas multicamadas também podem ser usados quando aplicações precisam acessar e usar dados de diferentes bancos de dados. Nesse caso, talvez você precise adicionar um servidor de integração ao sistema, o qual atuaria coletando os dados distribuídos e apresentando-os ao servidor de aplicação, como se tratasse de um único banco de dados. Como discuto na seção seguinte, arquiteturas de componentes distribuídos podem ser usadas para implementar sistemas cliente-servidor multicamadas.

Figura 18.10 Uma arquitetura de componentes distribuídos

Os sistemas cliente-servidor multicamadas que distribuem o processamento de aplicação entre vários servidores são inerentemente mais escaláveis do que as arquiteturas de duas camadas. O processamento de aplicação é, muitas vezes, a parte mais volátil do sistema e pode ser facilmente atualizado, pois está centralmente localizado. O processamento, em alguns casos, pode ser distribuído entre os servidores de lógica de aplicação e de gerenciamento de dados, gerando uma resposta mais rápida para as solicitações de clientes.

Os projetistas de arquiteturas cliente-servidor devem considerar vários fatores ao escolher a arquitetura de distribuição mais adequada. A Tabela 18.1 descreve situações em que as arquiteturas cliente-servidor podem ser adequadas.

18.3.4 Arquiteturas de componentes distribuídos

A Figura 18.5 mostra o processamento organizado em camadas, e cada camada de um sistema pode ser implementada como um servidor lógico separado. Esse modelo funciona bem para muitos tipos de aplicação. No entanto, ele limita a flexibilidade de projetistas de sistema, pois é necessário decidir quais serviços devem ser incluídos em cada camada. Na prática, contudo, não é sempre claro se um serviço é um serviço de gerenciamento de dados, um serviço de aplicação ou um serviço de banco de dados. Os projetistas também devem planejar para escalabilidade e, assim, fornecer alguns meios para que os servidores sejam replicados à medida que mais clientes são adicionados ao sistema.

Uma abordagem mais geral de projeto de sistemas distribuídos é projetar o sistema como um conjunto de serviços, sem tentar alocar esses serviços nas camadas do sistema. Cada serviço, ou grupo de serviços relacionados, é implementado usando um componente separado. Em uma arquitetura de componentes distribuídos (Figura 18.10), o sistema é organizado como um conjunto de componentes ou objetos interativos. Esses componentes fornecem uma interface para um conjunto de serviços que eles fornecem. Outros componentes chamam esses serviços através do *middleware*, usando chamadas de procedimento remoto ou chamadas de métodos.

Os sistemas de componentes distribuídos são dependentes do *middleware*, o qual gerencia as interações de componentes, reconcilia as diferenças entre os tipos de parâmetros passados entre componentes e fornece um conjunto de serviços comuns que os componentes de aplicação podem usar. CORBA (ORFALI et al., 1997) foi um dos primeiros exemplos de tal *middleware*, mas hoje em dia ele não é usado amplamente, pois tem sido suplantado por softwares proprietários como o Enterprise Java Beans (EJB) ou .NET.

Tabela 18.1 Uso de padrões de arquitetura cliente-servidor

Arquitetura	Aplicações
Arquitetura cliente-servidor de duas camadas com clientes-magros	Aplicações de sistema legado usadas quando a separação de gerenciamento de dados e de processamento de aplicação são impraticáveis. Os clientes podem acessá-las como serviços, conforme discutido na Seção 18.4. Aplicações computacionalmente intensivas como compiladores com pouco ou nenhum gerenciamento de dados. Aplicações intensivas de dados (navegação e consulta) com processamento de aplicações não intensivo. Navegar na Web é o exemplo mais comum de uma situação em que essa arquitetura é usada.
Arquitetura cliente-servidor de duas camadas com clientes-gordos	Aplicações em que o processamento de aplicação é fornecido por softwares de prateleira (por exemplo, o Microsoft Excel) no cliente. Aplicações em que é requerido processamento computacionalmente intensivo de dados (por exemplo, visualização de dados). Aplicações móveis em que a conectividade com a Internet não pode ser garantida. Algum processamento local usando informações armazenadas em banco de dados, portanto, é possível.
Arquitetura cliente-servidor multicamadas	Aplicações de grande porte com centenas ou milhares de clientes. Aplicações nas quais os dados e a aplicação são voláteis e integrados a dados de várias fontes.

Os benefícios de se usar um modelo de componentes distribuídos para implementação de sistemas distribuídos são os seguintes:

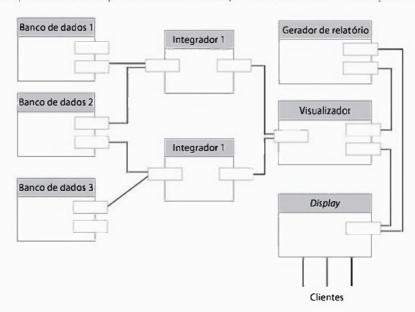
1. A permissão ao projetista de sistema de atrasar decisões sobre onde e como os serviços deverão ser prestados. Os componentes fornecedor de serviços podem executar em qualquer nó da rede. Não é necessário decidir previamente se um serviço é parte de uma camada de gerenciamento de dados, uma camada de aplicação etc.
2. É uma arquitetura de sistemas muito aberta, a qual permite a adição de novos recursos conforme necessário. Novos serviços de sistema podem ser adicionados facilmente sem grandes perturbações ao sistema existente.
3. O sistema é flexível e escalável. Novos componentes ou componentes replicados podem ser adicionados quando a carga sobre o sistema aumenta, sem interromper as outras partes do sistema.
4. É possível reconfigurar o sistema dinamicamente com componentes migrando através da rede conforme necessário. Isso pode ser importante onde estão flutuando padrões de demanda de serviços. Um componente fornecedor de serviços pode migrar para o mesmo processador como objetos requisitor de serviços, aumentando assim o desempenho do sistema.

Uma arquitetura de componentes distribuídos pode ser usada como um modelo lógico que permite estruturar e organizar o sistema. Nesse caso, poderá fornecer a funcionalidade de aplicação, unicamente, em termos de serviços e combinações de serviços. Depois, você define como fornecer esses serviços usando um conjunto de componentes distribuídos. Por exemplo, em uma aplicação de varejo pode haver componentes de aplicação interessados no controle de estoque, comunicações com clientes, pedidos de mercadorias, e assim por diante.

Os sistemas de mineração de dados são um bom exemplo de um tipo de sistema no qual uma arquitetura de componentes distribuídos é o melhor padrão de arquitetura para se usar. Um sistema de mineração de dados procura relacionamentos entre os dados que são armazenados em uma série de bancos de dados (Figura 18.11). Os sistemas de mineração de dados geralmente extraem informações de vários bancos de dados separados e realizam o processamento computacional intensivo e exibem seus resultados em gráficos.

Um exemplo de uma aplicação de mineração de dados pode ser um sistema para uma empresa de varejo que vende livros e mercadorias. O departamento de marketing quer encontrar relacionamentos entre as compras de mercadorias e livros de um cliente. Por exemplo, uma proporção relativamente elevada de pessoas que compram pizzas também podem comprar romances policiais. Com esse conhecimento, o negócio pode voltar-se para os clientes que fazem compras de mercadorias específicas com informações sobre novos romances, quando estes são publicados.

Nesse exemplo, cada banco de dados de vendas pode ser sintetizado como um componente distribuído com uma interface que fornece acesso somente para a leitura de seus dados. Os componentes integradores estão interessados em tipos específicos de relacionamentos que coletam informações de todos os bancos de dados para tentar deduzir os relacionamentos. Pode haver um componente integrador que se preocupa com variações sazonais dos bens vendidos e outro que se preocupa com os relacionamentos entre os diferentes tipos de mercadorias.

Figura 18.11 Uma arquitetura de componentes distribuídos para um sistema de mineração de dados

Os componentes visualizadores interagem com os componentes integradores para produzir uma visualização ou um relatório sobre os relacionamentos que forem descobertos. Por causa dos grandes volumes de dados que são manipulados, os componentes visualizadores normalmente apresentam seus resultados graficamente. Finalmente, um componente *display* pode ser responsável por entregar os modelos gráficos para clientes para a apresentação final.

Uma arquitetura de componentes distribuídos é apropriada para esse tipo de aplicação, ao invés de uma arquitetura em camadas, pois novos bancos de dados podem ser adicionados ao sistema sem grandes perturbações. Cada novo banco de dados é acessado adicionando-se outro componente distribuído. Os componentes de acesso ao banco de dados fornecem uma interface simplificada que controla o acesso aos dados. Os bancos de dados acessados podem residir em máquinas diferentes. A arquitetura também torna mais fácil minerar novos tipos de relacionamentos, adicionando novos componentes integradores.

As arquiteturas de componentes distribuídos sofrem de duas grandes desvantagens:

1. Elas são mais complexas para projetar que sistemas cliente-servidor. Os sistemas cliente-servidor multicamadas parecem ser uma forma bastante intuitiva de se pensar sobre os sistemas. Eles refletem muitas transações humanas em que as pessoas solicitam e recebem serviços de outras pessoas que se especializaram em fornecer tais serviços. Por outro lado, as arquiteturas de componentes distribuídos são mais difíceis para as pessoas visualizarem e compreenderem.
2. O *middleware* padronizado para sistemas de componentes distribuídos nunca foi aceito pela comunidade. Diferentes fornecedores, como a Microsoft e a Sun, desenvolveram *middlewares* diferentes e incompatíveis. Esses *middlewares* são complexos e a confiança neles aumenta a complexidade geral dos sistemas distribuídos.

Como resultado desses problemas, as arquiteturas orientadas a serviços (discutidas no Capítulo 19) estão substituindo as arquiteturas de componentes distribuídos em muitas situações. No entanto, os sistemas de componenes distribuídos têm benefícios de desempenho sobre os sistemas orientados a serviços. Geralmente, as comunicações RPC são mais rápidas do que a interação baseada em mensagens usada em sistemas orientados a serviços. As arquiteturas baseadas em componentes, portanto, são mais adequadas para sistemas com alta taxa de transferência em que um grande número de transações precisa ser processado rapidamente.

18.3.5 Arquiteturas ponto-a-ponto

O modelo de computação cliente-servidor discutido nas seções anteriores do capítulo faz uma distinção clara entre os servidores, que são provedores de serviços, e clientes, que são receptores de serviços. Geralmente, esse modelo leva a uma distribuição desigual de carga no sistema, em que os servidores trabalham mais que clientes. Isso pode levar as organizações a investirem muito na capacidade de servidor, enquanto existe capacidade de processamento não usada em centenas ou milhares de PCs usados para acessar os servidores de sistema.

Os sistemas ponto-a-ponto (p2p, do inglês *peer-to-peer*) são sistemas descentralizados em que os processamentos podem ser realizados por qualquer nó na rede. Em princípio, pelo menos, não existem distinções entre clientes e servidores. Em aplicações ponto-a-ponto, todo o sistema é projetado para aproveitar o poder computacional e o armazenamento disponível por meio de uma rede potencialmente enorme de computadores. As normas e protocolos que permitem a comunicação entre os nós são embutidas na própria aplicação, e cada nó deve executar uma cópia dessa aplicação.

As tecnologias ponto-a-ponto têm sido usadas, principalmente, para sistemas pessoais, e não de negócio (ORAM, 2001). Por exemplo, os sistemas de compartilhamento de arquivos com base em protocolos Gnutella e BitTorrent são usados para trocar arquivos de PCs. Os sistemas de mensagens instantâneas, como o ICQ e Jabber, fornecem comunicação direta entre os usuários sem um servidor intermediário. SETI@home é um projeto de longa duração para processar dados de radiotelescópios em PCs domésticos para procurar indícios de vida extraterrestre. Freenet é um banco de dados descentralizado que foi projetado para tornar mais fácil publicar informações anonimamente e para tornar mais difícil para as autoridades suprimirem essa informação. Serviços de voz sobre IP (VOIP), como o Skype, dependem da comunicação ponto-a-ponto entre as partes envolvidas na chamada telefônica ou conferência.

No entanto, os sistemas ponto-a-ponto também estão sendo usados pelas empresas para aproveitarem o poder em suas redes de PC (McDOUGALL, 2000). A Intel e a Boeing têm dois sistemas p2p implementados para aplicações computacionalmente intensivas. Elas aproveitam a capacidade de processamento não usada em computadores locais. Em vez de comprar hardwares caros, de alto desempenho, os processamentos de engenharia

podem ser executados durante a noite, quando os computadores *desktop* não são usados. As empresas também fazem uso extensivo de sistemas p2p comerciais, como sistemas de mensagens e VOIP.

É adequado usar um modelo de arquitetura ponto-a-ponto para um sistema em duas circunstâncias:

1. Quando o sistema é computacionalmente intensivo e é possível separar o processamento necessário para um grande número de computações independentes. Por exemplo, um sistema ponto-a-ponto que ofereça suporte computacional para a descoberta de novas drogas distribui computações que procuram possíveis tratamentos de câncer, analisando um elevado número de moléculas para ver se elas têm as características necessárias para suprimir o crescimento de cânceres. Cada molécula pode ser considerada isoladamente, então não existe nenhuma necessidade de comunicação entre os pontos no mesmo nível de sistema.
2. Sempre que o sistema envolver a troca de informações entre computadores individuais em uma rede e não for necessário que essas informações sejam armazenadas ou gerenciadas centralmente. Exemplos de tais aplicações incluem sistemas de compartilhamento de arquivos que permitem que os pontos troquem de arquivos localmente, como música e arquivos de vídeo e sistemas de telefone que oferecem suporte a comunicações de voz e vídeo entre computadores.

Em princípio, cada nó em uma rede p2p poderia estar ciente de todos os outros nós. Os nós poderiam conectar-se e trocar dados diretamente com qualquer outro nó da rede. Na prática, isso é certamente impossível, por isso os nós são organizados em 'localidades' com alguns nós atuando como pontes para outras localidades de nós. A Figura 18.12 mostra essa arquitetura p2p descentralizada.

Em uma arquitetura descentralizada, os nós da rede não são simplesmente elementos funcionais, mas também comutadores de comunicações que podem rotear dados e controlar os sinais de um nó para outro. Por exemplo, suponha que a Figura 18.12 represente um sistema de gerenciamento de documentos descentralizado. Esse sistema é usado por um consórcio de pesquisadores para compartilhar documentos e cada membro do consórcio mantém seu próprio repositório de documentos. No entanto, quando um documento é recuperado, o nó que o recupera também o disponibiliza para outros nós.

Se alguém precisa de um documento que é armazenado em algum lugar na rede, essa pessoa emite um comando de busca, que é enviado para os nós em sua 'localidade'. Esses nós verificam se eles têm o documento e, em caso afirmativo, devolvem o documento para o solicitante. Se não tiverem, eles roteam a pesquisa para outros nós. Portanto, se n1 emitir uma pesquisa para um documento que está armazenado no n10, essa pesquisa será roteada através dos nós n3, n6 e n9 a n10. Quando o documento é descoberto, o nó que possui o documento envia-o diretamente para o nó solicitante, fazendo uma conexão ponto-a-ponto.

Essa arquitetura descentralizada tem vantagens, pois é altamente redundante e, portanto, tolerante a defeitos e à desconexão de nós da rede. No entanto, as desvantagens são que muitos nós diferentes podem processar a mesma pesquisa e ocorrer um *overhead* significativo em comunicações de pontos replicadas.

Um modelo de arquitetura p2p alternativo, que parte de uma arquitetura p2p pura, é uma arquitetura semicentralizada em que, no âmbito da rede, um ou mais nós atuam como servidores para facilitar as comunicações entre os nós. Isso reduz o tráfego entre eles. A Figura 18.13 ilustra esse modelo.

Em uma arquitetura semicentralizada, a função do servidor (às vezes chamado superponto) é ajudar no estabelecimento de contato entre os pontos da rede ou na coordenação dos resultados de um processamento. Por exemplo, se a Figura 18.13 representa um sistema de mensagens instantâneas, então os nós de rede se comuni-

Figura 18.12 Uma arquitetura p2p descentralizada

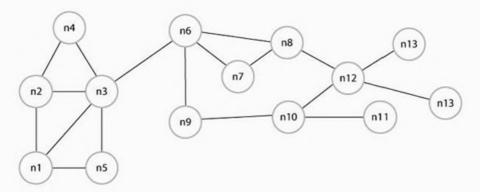
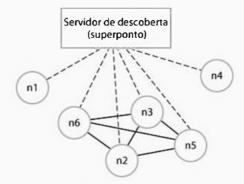



Figura 18.13 Uma arquitetura p2p semicentralizada

cam com o servidor (indicado por linhas tracejadas) para saber quais outros nós estão disponíveis. Uma vez que esses nós são descobertos, pode-se estabelecer a comunicação direta e a conexão com o servidor será desnecessária. Portanto, os nós n2, n3, n5 e n6 estão em comunicação direta.

Em um sistema computacional p2p em que uma computação de processador intensivo é distribuída por meio de um grande número de nós, é normal que alguns nós sejam superpontos. Seu papel é distribuir o trabalho para outros nós, conferir e verificar os resultados da computação.

As arquiteturas ponto-a-ponto permitem o uso eficiente da capacidade por meio de uma rede. No entanto, os principais problemas que têm inibido seu uso são as questões de proteção e confiança. A comunicação ponto-a-ponto envolve abrir seu computador para direcionar as interações com outros pontos, e isso significa que esses sistemas poderiam, potencialmente, acessar qualquer um de seus recursos. A fim de combater isso, você precisa organizar seu sistema para que esses recursos sejam protegidos. Caso esse processo seja feito incorretamente, o sistema poderá ficar inseguro.

Também podem ocorrer problemas quando os pontos em uma rede se comportam deliberadamente de forma maliciosa. Por exemplo, já houve casos em que empresas de música, acreditando que seus direitos autorais estavam sendo violados, 'envenenaram os pontos' disponíveis, deliberadamente. Quando o outro ponto baixa o que eles acham que é uma peça de música, o arquivo real entregue é um *malware*, que pode ser uma versão deliberadamente corrompida de música ou um aviso para o usuário da violação de direitos autorais.

18.4 Softwares como um serviço

Nas seções anteriores, discuti modelos de cliente-servidor e como a funcionalidade pode ser distribuída entre os clientes e os servidores. Para implementar um sistema cliente-servidor, talvez você precise instalar no computador do cliente um programa que se comunique com o servidor, implemente a funcionalidade do cliente e gerencie a interface de usuário. Por exemplo, um cliente de e-mail, como Outlook ou Mac Mail, fornece recursos de gerenciamento de correio em seu próprio computador. Isso evita o problema de alguns sistemas cliente-magro, nos quais todo o processamento é realizado no servidor.

No entanto, os problemas de *overhead* de servidor podem ser significativamente reduzidos, usando-se um *browser* moderno como o software de cliente. As tecnologias Web, como AJAX (HOLDENER, 2008), suportam o gerenciamento eficiente de apresentação de página Web e a computação local por meio de *scripts*. Isso significa que um *browser* pode ser configurado e usado como um cliente, com processamento local significativo. O software de aplicação pode ser pensado como um serviço remoto, que pode ser acessado de qualquer dispositivo que possa executar um *browser*-padrão. Exemplos bem conhecidos disso são sistemas de correio baseados na Web, como Yahoo!® e Gmail®, além de aplicações de escritório, como o Google® Docs.

Essa noção de SaaS envolve a hospedagem remota do software e fornece acesso a ele através da Internet. Os elementos-chave do SaaS são os seguintes:

1. O software é implantado em um servidor (ou, mais comumente, vários servidores) e é acessado por meio de um *browser* de Web. Ele não é implantado em um PC local.
2. O software é de propriedade e gerido por um fornecedor de software, e não pelas organizações que usam o software.

3. Os usuários podem pagar para o software de acordo com a quantidade de uso que fazem dele ou por meio de uma assinatura anual ou mensal. Às vezes, o software tem o uso liberado, mas os usuários devem concordar em aceitar anúncios que financiem o serviço de software.

Para usuários de software, o benefício do SaaS é que os custos de gerenciamento de software são transferidos para o provedor. O provedor é responsável pela correção de *bugs* e instalação das atualizações de software, pelas alterações na plataforma de sistema operacional e pela garantia de que a capacidade do hardware possa atender à demanda. Os custos de gerenciamento de licenças de software são zero. Se alguém tiver vários computadores, não existe necessidade de licença de software para todos eles. Se uma aplicação de software é usada apenas ocasionalmente, o modelo 'pague pelo uso' (*pay-per-use*) pode ser mais barato do que comprar uma aplicação. O software pode ser acessado de dispositivos móveis, como *smart phones*, de qualquer lugar do mundo.

Naturalmente, esse modelo de fornecimento de software tem algumas desvantagens. O principal problema talvez seja os custos de transferência de dados para o serviço remoto. A transferência de dados ocorre em velocidade de rede e, então, transferir uma grande quantidade de dados leva muito tempo. Você também pode ter de pagar o provedor de serviços de acordo com o montante transferido. Outros problemas são a falta de controle sobre a evolução de software (o provedor pode alterar o software quando desejar), além de problemas com leis e regulamentos. Muitos países têm leis que regem o armazenamento, o gerenciamento, a preservação e a acessibilidade de dados, e mover os dados para um serviço remoto pode violar tais leis.

A noção de SaaS e as arquiteturas orientadas a serviços (SOAs), discutidas no Capítulo 19, relacionam-se, obviamente, mas não são as mesmas:

1. O SaaS é uma forma de fornecer funcionalidade em um servidor remoto com acesso de clientes por meio de um *browser* de Web. O servidor mantém os dados e o estado do usuário durante uma sessão de interação. Geralmente, as transações são longas (por exemplo, edição de um documento).
2. A SOA é uma abordagem para a estruturação de um sistema de software como um conjunto de serviços separados, sem estado. Estes podem ser fornecidos por vários provedores e podem ser distribuídos. Normalmente, tratam-se de transações curtas, em que um serviço é chamado, faz alguma coisa e, em seguida, retorna um resultado.

O SaaS é uma maneira de entregar a funcionalidade de aplicação para os usuários, enquanto a SOA é uma tecnologia de implementação para sistemas de aplicações. A funcionalidade implementada pelo uso da SOA precisa aparecer para usuários como serviços. Da mesma forma, os serviços de usuário não precisam ser implementados pelo uso da SOA. No entanto, se o SaaS é implemento usando a SOA, torna-se possível para aplicações usarem APIs de serviço para acessar a funcionalidade de outras aplicações. Em seguida, estas podem ser integradas em sistemas mais complexos; são chamados *mashups* e representam outra abordagem para reúso de software e desenvolvimento rápido de software.

De uma perspectiva de desenvolvimento de software, o processo de desenvolvimento de serviços tem muito em comum com outros tipos de desenvolvimento de software. No entanto, a construção de serviços normalmente não é conduzida pelos requisitos do usuário, mas por suposições do provedor de serviços sobre o que os usuários precisam. Portanto, o software precisa ser capaz de evoluir rapidamente depois que o provedor obtenha *feedback* dos usuários sobre seus requisitos. Portanto, o desenvolvimento ágil com entrega incremental é uma abordagem comumente usada para os softwares que devem ser implantados como serviços.

Ao implementar o SaaS, você precisa considerar que pode haver usuários do software de várias organizações diferentes. São três os fatores que precisam ser considerados:

1. *Configurabilidade*. Como você configura o software para as necessidades específicas de cada organização?
2. *Multilocação*. Como você apresenta para cada usuário do software a impressão de que eles estão trabalhando com sua própria cópia do sistema enquanto, e ao mesmo tempo, fazem uso eficiente dos recursos de sistema?
3. *Escalabilidade*. Como você projeta o sistema para que ele possa ser dimensionado a fim de acomodar um número imprevisível de usuários?

A noção de arquiteturas de linha de produtos, discutida no Capítulo 16, é uma forma de configurar o software para usuários que possuem sobreposição, mas requisitos não idênticos. Você começa com um sistema genérico e o adapta de acordo com os requisitos específicos de cada usuário.

No entanto, isso não funciona para SaaS, pois significaria implantar uma cópia diferente do serviço para cada organização que usa o software. Em vez disso, você precisa projetar a configurabilidade no sistema e fornecer uma interface de configuração que permita aos usuários especificarem suas preferências. Em seguida, você usa esses dados para ajustar o comportamento do software, dinamicamente, enquanto ele é usado. Os recursos de configuração podem permitir:

1. *Gerenciamento de marcas*, em que aos usuários de cada organização são apresentados com uma interface que reflete sua própria organização.
2. *Regras de negócios e fluxos de trabalho*, em que cada organização define suas próprias regras para regerem o uso do serviço e seus dados.
3. *Extensões de banco de dados*, em que cada organização define como o modelo de dados do serviço genérico é ampliado para atender a suas necessidades específicas.
4. *Controle de acesso*, em que os clientes do serviço criam contas individuais para sua equipe e definem os recursos e funções acessíveis para cada um de seus usuários.

A Figura 18.14 ilustra essa situação. Esse diagrama mostra cinco usuários do serviço de aplicação, os quais trabalham para três clientes diferentes do provedor de serviços. Os usuários interagem com o serviço por meio de um perfil de clientes que define a configuração de serviço para seu empregador.

A multilocação é uma situação em que muitos diferentes usuários acessam o mesmo sistema e a arquitetura do sistema é definida para permitir o compartilhamento eficiente dos recursos de sistema. No entanto, para cada usuário deve parecer que ele tem o uso exclusivo do sistema. A multilocação envolve projetar o sistema para que haja uma separação absoluta entre sua funcionalidade e seus dados. Portanto, você deve projetar o sistema para que todas as operações sejam sem estado. Os dados devem ser fornecidos pelo cliente ou devem estar disponíveis em um sistema de armazenamento ou banco de dados que possa ser acessado a partir de qualquer instância do sistema. Os bancos de dados relacionais não são ideais para fornecimento de multilocação e grandes provedores de serviços, como Google®, implementaram um banco de dados simples para os dados de usuários.

Um problema específico em sistemas de multilocação é o gerenciamento de dados. A maneira mais simples de fornecer o gerenciamento de dados é garantir que cada cliente tenha seu próprio banco de dados, para que eles possam usá-lo e configurá-lo como quiserem. No entanto, isso requer que o provedor de serviços mantenha muitas instâncias de banco de dados diferentes (um por cliente) para disponibilizá-las sob demanda. Em termos da capacidade de servidor, esse procedimento é ineficiente, além de aumentar o custo global do serviço.

Como alternativa, o provedor de serviços pode usar um único banco de dados com diferentes usuários virtualmente isolados dentro desse banco de dados. Isso é ilustrado na Figura 18.15, na qual se pode ver que entradas de banco de dados também têm um 'identificador de locatário', que liga essas entradas a usuários específicos. Usando visões de banco de dados, você pode extrair as entradas para cada cliente de serviço e, assim, apresentar os usuários desse cliente com um banco de dados virtual e pessoal, o que pode ser estendido para atender às necessidades específicas do cliente, usando os recursos de configuração discutidos anteriormente.

Figura 18.14 Configuração de um sistema de software oferecido como um serviço

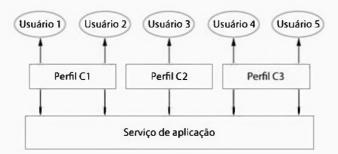



Figura 18.15 Um banco de dados de multilocações

Locatário	Chave	Nome	Endereço
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

A escalabilidade é a capacidade do sistema de lidar com o aumento do número de usuários sem reduzir o QoS global que é entregue a qualquer usuário. Geralmente, ao considerar a escalabilidade no contexto do SaaS, você está considerando 'escalamento para fora' ao invés de 'escalamento para cima'. Lembre-se de que 'escalamento para fora' significa adicionar servidores adicionais e, assim, também, aumentar o número de transações que podem ser processadas em paralelo. A escalabilidade é um tópico complexo, o qual não discuto em detalhes mas algumas diretrizes gerais para a implementação de softwares escaláveis são:

1. Desenvolva aplicações em que cada componente é implementado como um serviço simples, sem estado, o qual pode ser executado em qualquer servidor. Portanto, no decurso de uma única transação, um usuário pode interagir com instâncias do mesmo serviço em execução em diversos servidores.
2. Projete o sistema usando a interação assíncrona para que a aplicação não tenha de esperar o resultado de uma interação (por exemplo, uma solicitação de leitura). Isso permite que a aplicação continue a realizar um trabalho útil enquanto está aguardando a interação terminar.
3. Gerencie recursos, como conexões de rede e banco de dados, como um *pool*, para que nenhum servidor específico corra o risco de ficar sem recursos.
4. Projete seu banco de dados para permitir o bloqueio de baixa granularidade. Ou seja, não bloqueie registros inteiros no banco de dados quando apenas parte de um registro está em uso.

A noção de SaaS é uma grande mudança de paradigma para a computação distribuída. Ao invés de uma organização que hospeda várias aplicações em seus servidores, SaaS permite que essas aplicações sejam fornecidas externamente, por diferentes fornecedores. Estamos no meio da transição de um modelo para outro, e é provável que, no futuro, esse processo tenha um efeito significativo sobre a engenharia de sistemas de software corporativos.

PONTOS IMPORTANTES

- Os benefícios de sistemas distribuídos são que eles podem ser dimensionados para lidar com o aumento da demanda, podem continuar a fornecer serviços de usuário (mesmo que algumas partes do sistema falhem) e habilitam o compartilhamento de recursos.
- Algumas questões importantes no projeto de sistemas distribuídos incluem transparência, abertura, escalabilidade, proteção, qualidade de serviço e gerenciamento de falhas.
- Os sistemas cliente-servidor são sistemas distribuídos em que o sistema está estruturado em camadas, com a camada de apresentação implementada em um computador cliente. Os servidores fornecem serviços de gerenciamento de dados, de aplicações e de banco de dados.
- Os sistemas cliente-servidor podem ter várias camadas, com diferentes camadas do sistema distribuídas em computadores diferentes.
- Os padrões de arquitetura para sistemas distribuídos incluem arquiteturas mestre-escravo, arquiteturas cliente--servidor de duas camadas e de múltiplas camadas, arquiteturas de componentes distribuídos e arquiteturas ponto-a-ponto.
- Os componentes de sistemas distribuídos requerem *middleware* para lidar com as comunicações de componentes e para permitir que componentes sejam adicionados e removidos do sistema.
- As arquiteturas ponto-a-ponto são arquiteturas descentralizadas em que não existe distinção entre clientes e servidores. As computações podem ser distribuídas ao longo de muitos sistemas, em organizações diferentes.
- O software como um serviço é uma maneira de implantar aplicações como sistemas cliente-magro-servidor, em que o cliente é um *browser* de Web.

LEITURA COMPLEMENTAR

'Middleware: A model for distributed systems services'. Embora um pouco ultrapassado em partes, esse é um excelente artigo, que oferece uma visão geral e resume o papel do *middleware* em sistemas distribuídos, além de discutir a gama de serviços de *middleware* que podem ser fornecidos. (BERNSTEIN, P. A. *Comm. ACM*, v. 39, n. 2, fev. 1996.) Disponível em: <http://dx.doi.org/10.1145/230798.230809>.

Peer-to-Peer: Harnessing the Power of Disruptive Technologies. Embora esse livro não tenha muita informação sobre arquiteturas p2p, ele é uma excelente introdução à computação p2p e discute a organização e a aborda-

gem usadas em vários sistemas p2p. (ORAM, R. (Org.). *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly and Associates Inc., 2001.)

'Turning software into a service'. Um artigo com uma boa visão geral que discute os princípios da computação orientada a serviços. Ao contrário de muitos artigos sobre o tema, esse não esconde esses princípios por trás de uma discussão sobre os padrões envolvidos. (TURNER, M.; BUDGEN, D.; BRERETON, P. *IEEE Computer*, v. 36, n. 10, out. 2003.) Disponível em: <http://dx.doi.org/10.1109/MC.2003.1236470>.

Distributed Systems: Principles and Paradigms, 2nd edition. Um livro-texto abrangente que aborda todos os aspectos do projeto e implementação de sistemas distribuídos. No entanto, ele não inclui muitas discussões sobre o paradigma orientado a serviços. (TANENBAUM, A. S.; VAN STEEN, M. *Distributed Systems: Principles and Paradigms.* 2. ed. Addison-Wesley, 2007.)

'Software as a Service; The Spark that will Change Software Engineering'. Um pequeno artigo que argumenta que o advento do SaaS vai impulsionar todo o desenvolvimento de software para um modelo iterativo. (GOTH, G. *Distributed Systems Online*, v. 9, n. 7, jul. 2008.) Disponível em: <http://dx.doi.org/10.1109/MDSO.2008.21>.

EXERCÍCIOS

18.1 O que você entende por 'escalabilidade'? Discuta as diferenças entre 'escalabilidade para cima' e 'escalabilidade para fora' e explique quando essas diferentes abordagens para a escalabilidade podem ser usadas.

18.2 Explique por que sistemas de software distribuídos são mais complexos do que os sistemas de software centralizados, em que toda a funcionalidade de sistema é implementada em um único computador.

18.3 Usando um exemplo de uma chamada de procedimento remoto, explique como o *middleware* coordena a interação entre os computadores em um sistema distribuído.

18.4 Qual é a diferença fundamental entre a abordagem cliente-gordo e a abordagem cliente-magro para as arquiteturas de sistemas cliente-servidor?

18.5 Foi solicitada a criação de um sistema protegido que requer autorização e autenticação forte. O sistema deve ser projetado para que as comunicações entre as partes do sistema não possam ser interceptadas e lidas por um invasor. Sugira a arquitetura cliente-servidor mais adequada para esse sistema e, justificando sua resposta, proponha como a funcionalidade deve ser distribuída entre os sistemas cliente e servidor.

18.6 Seu cliente quer desenvolver um sistema de informações de estoque em que os revendedores possam acessar informações sobre empresas e avaliar diferentes cenários de investimento por meio de um sistema de simulação. Cada revendedor(a) usa essa simulação de forma diferente, de acordo com sua experiência e o tipo de estoque em questão. Sugira uma arquitetura cliente-servidor para esse sistema que mostre onde se encontra a funcionalidade. Justifique o modelo do sistema cliente-servidor que você selecionou.

18.7 Usando uma abordagem de componentes distribuídos, proponha uma arquitetura para um sistema nacional de reserva para teatro. Os usuários podem verificar a disponibilidade e reservar os assentos em um grupo de teatros. O sistema deve aceitar a devolução de bilhetes, assim, as pessoas podem devolver seus bilhetes para vendas de última hora para outros clientes.

18.8 Apresente duas vantagens e duas desvantagens de arquiteturas ponto-a-ponto descentralizadas e semicentralizadas.

18.9 Explique por que implantar software como um serviço pode reduzir os custos de suporte de TI para uma empresa. Quais custos adicionais podem surgir caso esse modelo de implantação seja usado?

18.10 Sua empresa pretende parar de usar aplicações de *desktop* para acessar a mesma funcionalidade remotamente, como serviços. Identifique três riscos que podem surgir e dê sugestões de como diminuir esses riscos.

REFERÊNCIAS

BERNSTEIN, P. A. Middleware: A Model for Distributed System Services. *Comm. ACM*, v. 39, n. 2, 1996, p. 86-97.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed Systems*: Concepts and Design, 4.ed. Harlow, Reino Unido: Addison-Wesley, 2005.

HOLDENER, A. T. *Ajax*: The Definitive Guide. Sebastopol, Calif.: O'Reilly and Associates, 2008.

McDOUGALL, P. The Power of Peer-To-Peer. *Information Week*, 28 ago. 2000.

NEUMAN, B. C. Scale in Distributed Systems. In: CASAVANT, T.; SINGAL, M. (Orgs.). *Readings in Distributed Computing Systems*. Los Alamitos, Calif.: IEEE Computer Society Press, 1994.

ORAM, A. Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology, 2001.

ORFALI, R.; HARKEY, D. *Client/server Programming with Java and CORBA*. Nova York: John Wiley & Sons, 1998.

ORFALI, R.; HARKEY, D.; EDWARDS, J. *Instant CORBA*. Chichester, Reino Unido: John Wiley & Sons, 1997.

POPE, A. *The CORBA Reference Guide*: Understanding the Common Request Broker Architecture. Boston: Addison--Wesley, 1997.

TANENBAUM, A. S.; VAN STEEN, M. *Distributed Systems*: Principles and Paradigms, 2.ed. Upper Saddle River, NJ: Prentice Hall, 2007.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 **19** 20 21 22 23 24 25 26

Arquitetura orientada a serviços

Objetivos

O objetivo deste capítulo é apresentar a arquitetura de software orientada a serviços como uma forma de criação de aplicações distribuídas usando *web services*. Com a leitura deste capítulo, você:

- compreenderá as noções básicas de *web service*, padrões de *web services* e arquitetura orientada a serviços;
- entenderá o processo da engenharia de serviços que se destina a produzir *web services* reusáveis;
- conhecerá a noção de composição de serviços como um meio de desenvolvimento de aplicações orientadas a serviços;
- entenderá como modelos de processo de negócios podem ser usados como uma base para o projeto de sistemas orientados a serviços.

Conteúdo

Na década de 1990, o desenvolvimento da Web revolucionou a troca de informações organizacionais. Os computadores de clientes poderiam acessar informações em servidores remotos fora de suas próprias organizações. No entanto, o acesso era exclusivamente por meio de um *browser* de Web e o acesso direto à informação por outros programas não era prático. Isso significava que não eram possíveis conexões oportunistas entre servidores nas quais, por exemplo, um programa consultava um número de catálogos de fornecedores diferentes.

Para contornar esse problema, foi proposta a ideia de um *web service*. Usando um *web service*, as organizações que desejam disponibilizar suas informações para outros programas podem fazê-lo definindo e publicando uma interface de *web service*. Essa interface define os dados disponíveis e como eles podem ser acessados. Geralmente, um *web service* é uma representação-padrão para algum recurso computacional ou de informações que pode ser usado por outros programas, os quais podem ser recursos de informações, como um catálogo de peças, recursos de computador, tais como um processador especializado, ou recursos de armazenamento. Por exemplo, pode ser implementado um serviço de arquivo que armazene permanentemente e de maneira confiável os dados organizacionais que, por lei, precisam ser mantidos por muitos anos.

Um *web service* é uma instância de uma ideia mais geral de um serviço, que é definido (LOVELOCK et al., 1996) como:

> *um ato ou desempenho oferecido de uma parte para outra. Embora o processo possa ser vinculado a um produto físico, o desempenho é essencialmente intangível e normalmente não resulta na posse de qualquer um dos fatores de produção.*

Figura 19.1 Arquitetura orientada a serviço

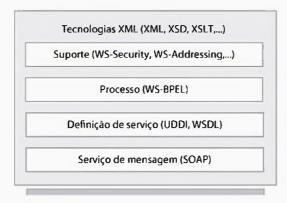
Portanto, a essência de um serviço é que o fornecimento de serviço é independente da aplicação que o usa (TURNER et al., 2003). Os provedores de serviços podem desenvolver serviços especializados e oferecê-los para uma variedade de usuários de serviço de diferentes organizações.

As arquiteturas orientadas a serviços (SOA, do inglês *service-oriented architectures*) são uma forma de desenvolvimento de sistemas distribuídos em que os componentes de sistema são serviços autônomos, executando em computadores geograficamente distribuídos. Protocolos-padrão baseados em XML, SOAP e WSDL foram projetados para oferecer suporte à comunicação de serviço e à troca de informações. Consequentemente, os serviços são plataforma e implementação independentes de linguagem. Os sistemas de software podem ser construídos pela composição de serviços locais e serviços externos de provedores diferentes, com interação perfeita entre os serviços no sistema.

A Figura 19.1 sintetiza a ideia de uma SOA. Os provedores de serviços projetam e implementam serviços e especificam a interface para esses serviços. Eles também publicam informações sobre esses serviços em um registro acessível. Os solicitantes (às vezes chamados clientes de serviço) de serviço que desejam usar um serviço descobrem a especificação desse serviço e localizam o provedor de serviço. Em seguida, eles podem ligar sua aplicação a esse serviço específico e comunicar-se com ele, usando protocolos de serviço-padrão.

Desde o início, houve um processo de padronização ativo para SOA, trabalhando ao lado de evoluções técnicas. Todas as principais empresas de hardware e software estão comprometidas com essas normas. Como resultado, a SOA não sofreu as incompatibilidades que costumam surgir com as inovações técnicas, em que diferentes fornecedores mantêm sua versão proprietária da tecnologia. A Figura 19.2 mostra a pilha de normas fundamentais criadas para oferecer suporte aos *web services*. Devido a essa padronização precoce, os problemas, como os modelos múltiplos incompatíveis de componentes em CBSE, discutidos no Capítulo 17, não surgiram no desenvolvimento de sistemas orientados a serviços.

Os protocolos de *web services* cobrem todos os aspectos das SOA, desde os mecanismos básicos para troca de informações de serviço (SOAP) até os padrões de linguagem da programação (WS-BPEL). Esses padrões são todos baseados em XML, uma notação legível por máquina e humanos que permite a definição de dados estruturados, em que o texto é marcado com um identificador significativo. XML tem uma gama de tecnologias, como XSD para definição de esquemas, que são usadas para estender e manipular descrições de XML. Erl (2004) fornece um bom resumo de tecnologias XML e seu papel nos *web services*.

Figura 19.2 Padrões de *web services*

Resumidamente, os principais padrões para SOA de Web são:

1. *SOAP*. Esse é um padrão de trocas de mensagens que oferece suporte à comunicação entre os serviços. Ele define os componentes essenciais e opcionais das mensagens passadas entre serviços.
2. *WSDL*. A linguagem de definição de *web service* (do inglês *Web Service Definition Language*) é um padrão para a definição de interface de serviço. Define como as operações de serviço (nomes de operação, parâmetros e seus tipos) e associações de serviço devem ser definidas.
3. *WS-BPEL*. Esse é um padrão para uma linguagem de *workflow*, que é usada para definir programas de processo que envolvem vários serviços diferentes. Na Seção 19.3, discuto sobre a noção de programas de processo.

Também foi proposto um padrão de descoberta de serviços UDDI, mas ele não foi amplamente adotado. O padrão UDDI (do inglês *Universal Description, Discovery and Integration*) define os componentes de uma especificação de serviço, que pode ser usada para descobrir a existência do serviço. Esses componentes incluem informações sobre o provedor de serviço, os serviços fornecidos, o local da descrição WSDL da interface de serviço e informações sobre os relacionamentos de negócios. A intenção era que esse padrão permitisse que as empresas configurassem registros com descrições de UDDI definindo os serviços oferecidos por eles.

Várias empresas, tais como a Microsoft, configuraram registros UDDI nos primeiros anos do século XXI, mas agora todos eles estão fechados. Melhorias na tecnologia de mecanismo de pesquisa tornaram-se redundantes. A descoberta de serviços usando um mecanismo de pesquisa-padrão para procurar descrições WSDL devidamente comentadas é, atualmente, a abordagem preferida para descobrir serviços externos.

Os principais padrões SOA são suportados por uma gama de padrões de suporte que se concentram nos aspectos mais especializados da SOA. Existe um grande número de padrões de suporte que se destinam a apoiar a SOA em diferentes aplicações empresariais. Alguns exemplos dessas normas incluem:

1. *WS-Reliable Messaging*, um padrão para troca de mensagens que garante que elas serão entregues uma vez e apenas uma vez.
2. *WS-Security*, um conjunto de padrões que apoiam a proteção de *web services*, incluindo padrões que especificam a definição de políticas de proteção e padrões que cobrem o uso de assinaturas digitais.
3. *WS-Addressing*, que define como as informações de endereço devem ser representadas em uma mensagem SOAP.
4. *WS-Transactions*, que define como as transações através de serviços distribuídos devem ser coordenadas.

Os padrões de *web services* são um tópico enorme e neste trabalho eu não tenho espaço para discuti-los. Recomendo os livros de Erl (2004; 2005) para uma visão geral desses padrões. Suas descrições detalhadas também estão disponíveis na Web como documentos públicos.

Os atuais padrões de *web services* têm sido criticados como padrões 'pesados', muito gerais e ineficientes. Implementar esses padrões requer uma quantidade considerável de processamento para criar, transmitir e interpretar as mensagens XML associadas. Por essa razão, algumas organizações, como a Amazon, usam uma abordagem mais simples e mais eficiente para comunicação de serviços, usando os chamados serviços RESTful (RICHARDSON e RUBY, 2007). A abordagem RESTful oferece suporte à interação eficiente de serviço, mas não oferece suporte a recursos de nível empresarial, como a WS-Reliability e WS-Transactions. Pautasso et al. (2008) comparam a abordagem RESTful com *web services* padronizados.

A construção de aplicações baseadas em serviços permite que empresas e outras organizações cooperem e façam uso das funções de negócios umas das outras. Assim, sistemas que envolvem muitas trocas de informações pelas fronteiras das empresas, como sistemas de cadeia de suprimentos, em que uma empresa compra bens de outra, podem ser facilmente automatizados. As aplicações baseadas em serviço podem ser construídas por meio da ligação de serviços de diversos fornecedores usando uma linguagem de programação padrão ou uma linguagem de *workflow* especializada, conforme discutido na Seção 19.3.

As SOA são arquiteturas menos rígidas, nas quais as ligações de serviços podem mudar durante a execução. Isso significa que uma versão diferente, mas equivalente, do serviço, pode ser executada em diferentes momentos. Alguns sistemas serão construídos exclusivamente com o uso de *web services*, e outros misturarão *web services* com componentes desenvolvidos localmente. Para ilustrar como as aplicações que usam uma mistura de componentes e serviços podem ser organizadas, considere o seguinte cenário:

> *Um sistema de informações em um carro fornece aos motoristas informações sobre clima, condições de tráfego da estrada, informações locais, e assim por diante. Ele é ligado ao rádio do carro para que a informação seja entregue como um sinal em um canal de rádio específico. O carro é equipado com receptores GPS para descobrir sua posição, e, com base nessa posição, o sistema acessa uma gama de serviços de informação. Em seguida, as informações podem ser entregues na linguagem especificada pelo motorista.*

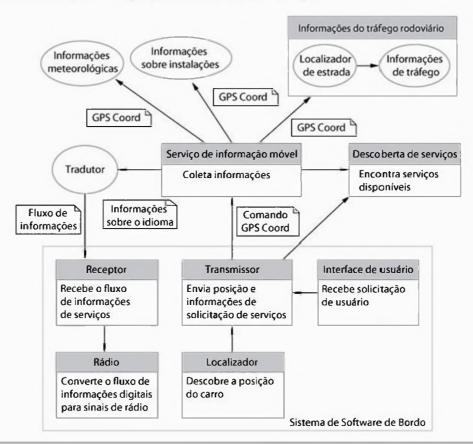
A Figura 19.3 ilustra uma organização possível para esse sistema. O software de bordo inclui cinco módulos, os quais lidam com comunicações com o motorista, com um receptor GPS que relata a posição do carro e com o rádio do carro. Os módulos Transmissor e Receptor lidam com todas as comunicações com os serviços externos.

O carro comunica-se com um serviço móvel externo de informação que agrega informações de uma variedade de outros serviços, fornecendo informações sobre clima, tráfego e instalações locais. Provedores diferentes em diferentes lugares oferecem esses serviços, e o sistema de bordo usa um serviço de descoberta para localizar serviços de informações adequadas e se ligarem a eles. O serviço de descoberta também é usado pelo serviço de informação móvel para conectar os serviços adequados sobre clima, tráfego e recursos. Serviços trocam mensagens SOAP que incluem informações sobre a posição do GPS usada pelos serviços para selecionar as informações apropriadas. A informação agregada é enviada para o carro por meio de um serviço que converte essas informações na linguagem preferida do motorista.

Esse exemplo ilustra uma das principais vantagens da abordagem orientada a serviços. Não é necessário decidir quando o sistema é programado ou implantado, qual provedor de serviço deve ser usado ou quais serviços específicos devem ser acessados. Conforme o carro se move, o software de bordo usa o serviço de descoberta de serviços para encontrar o serviço de informações mais adequado e os vincula. Por causa do uso de um serviço de tradução, ele pode mover-se além das fronteiras e, portanto, disponibilizar informações locais para as pessoas que não falam a língua local.

Uma abordagem orientada a serviços para a engenharia de software é um novo paradigma de engenharia de software que é, na minha opinião, um desenvolvimento tão importante quanto a engenharia de software orientada a objetos. Essa mudança de paradigma será acelerada pelo desenvolvimento de 'computação em nuvem' (CARR, 2009), em que os serviços são oferecidos em uma infraestrutura de computação pública hospedada por grandes fornecedores, como Google e Amazon. Isso teve e continuará a ter efeitos profundos sobre os produtos de sistemas e processos de negócios. Newcomer e Lomow (2005), em seu livro sobre SOA, resumem o potencial das abordagens orientada a serviços:

> *Impulsionada pela convergência de tecnologias-chave e a adoção universal de* web services, *a empresa orientada a serviços promete melhorar significativamente a agilidade empresarial, a velocidade da inserção de novos produtos e serviços no mercado, reduzir custos e melhorar a eficiência operacional.*

Figura 19.3 Um sistema de informações de bordo baseado em seviços

Ainda estamos em uma fase relativamente precoce do desenvolvimento de aplicações orientadas a serviços acessados pela Web. No entanto, estamos vendo grandes mudanças no modo como os softwares são implementados e implantados, com o surgimento de sistemas como o Google Apps e Salesforce.com. As abordagens orientadas a serviços, em ambos os níveis — de aplicação e de implementação — significam que a Web está evoluindo de um armazém de informações para uma plataforma de implementação de sistemas.

19.1 Serviços como componentes reusáveis

No Capítulo 17, eu apresentei a engenharia de software baseada em componentes (CBSE), na qual os sistemas de software são construídos pela composição de componentes de software baseados em um modelo-padrão de componentes. Os serviços são um desenvolvimento natural dos componentes de software em que o modelo de componente é, em essência, um conjunto de padrões associados com *web services*. Um serviço, portanto, pode ser definido como:

> *Um componente de software de baixo acoplamento, reusável, que encapsula funcionalidade discreta, que pode ser distribuída e acessada por meio de programas. Um* web service *é um serviço acessado usando protocolos baseados em padrões da Internet e em XML.*

Uma distinção fundamental entre um serviço e um componente de software, conforme definido na CBSE, é que os serviços devem ser independentes e fracamente acoplados; ou seja, eles sempre devem operar da mesma maneira, independentemente de seu ambiente de execução. Sua interface é uma interface 'provides' que permite o acesso à funcionalidade de serviço. Os serviços são projetados para serem independentes e podem ser usados em contextos diferentes. Portanto, eles não têm uma interface 'requires' que, em CBSE, define os outros componentes do sistema que devem estar presentes.

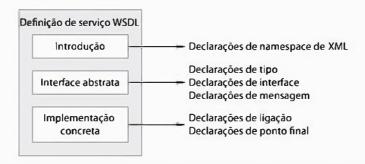
Os serviços se comunicam por meio de troca de mensagens, expressas em XML, e essas mensagens são distribuídas usando protocolos-padrão de transporte de Internet, como HTTP e TCP/IP. Na Seção 18.1.1, eu discuti essa abordagem baseada em mensagens para comunicação de componentes. Um serviço define o que precisa de outro serviço, definindo seus requisitos em uma mensagem e enviando-a a esse serviço. O serviço de recepção analisa a mensagem, efetua o processamento e, depois, envia uma resposta, como uma mensagem, para os serviços solicitantes. Esse serviço analisa a resposta para extrair as informações necessárias. Ao contrário dos componentes de software, os serviços não usam chamadas de procedimentos ou de métodos remotos para acessar a funcionalidade associada a outros serviços.

Quando você pretende usar um *web service*, precisa saber onde se encontra o serviço (sua URI) e os detalhes de sua interface. Estes são descritos em uma descrição de serviço expressa em uma linguagem baseada em XML, chamada WSDL. A especificação WSDL define três aspectos de um *web service*: o que faz o serviço, como ele se comunica e onde o encontrar:

1. O tópico 'o que' de um documento WSDL, denominado interface, especifica quais operações o serviço suporta e define o formato das mensagens que são enviadas e recebidas pelo serviço.
2. O 'como' de um documento WSDL, denominado ligação, mapeia a interface abstrata para um conjunto concreto de protocolos. A ligação especifica os detalhes técnicos de como se comunicar com um *web service*.
3. A parte 'onde' de um documento WSDL descreve o local da implementação de um *web service* específico (seu ponto final).

O modelo conceitual WSDL (Figura 19.4) mostra os elementos de uma descrição de serviço. Cada um desses é expresso em XML e pode ser fornecido em arquivos separados. Essas partes são:

1. Uma parte introdutória que geralmente define os *namespaces* de XML usados e que pode incluir uma seção de documentação, fornecendo informações adicionais sobre o serviço.
2. Uma descrição opcional dos tipos usados em mensagens trocadas pelo serviço.
3. Uma descrição da interface de serviço; ou seja, as operações que fornecem o serviço para outros serviços ou usuários.
4. Uma descrição das mensagens de entrada e saída processadas pelo serviço.
5. Uma descrição da ligação usada pelo serviço (ou seja, o protocolo de mensagens que será usado para enviar, receber mensagens). O *default* é SOAP, mas outras ligações também podem ser especificadas. A ligação define como as mensagens de entrada e saída associadas ao serviço devem ser empacotadas em uma mensagem e

Figura 19.4 A organização de uma especificação WSDL

especifica os protocolos de comunicação usados. A ligação também pode especificar como são incluídas as informações de apoio, como as credenciais de proteção ou identificadores de transação.

6. Uma especificação de ponto final que é o local físico do serviço, expresso como um identificador de recurso uniforme (URI, do inglês *Uniform Resource Identifier*) — o endereço de um recurso que pode ser acessado pela Internet.

Descrições completas de serviço, escritas em XML, são longas, detalhadas e tediosas de ler. Geralmente, elas incluem definições de *namespaces* de XML, que são qualificadores para nomes. Um identificador de *namespace* pode preceder qualquer identificador usado na descrição XML, tornando possível distinguir entre identificadores com o mesmo nome definido em diferentes partes de uma descrição XML. Você não precisa entender os detalhes de *namespaces* para entender os exemplos dados aqui; você só precisa saber que os nomes podem ser prefixados com um identificador de *namespace* e que o *namespace: name pair* deve ser único.

Atualmente, as especificações WSDL raramente são escritas à mão e a maioria das informações em uma especificação pode ser gerada automaticamente. Você não precisa conhecer os detalhes de uma especificação para compreender os princípios da WSDL. Dessa forma, eu me concentro na descrição da interface abstrata. Essa é a parte de uma especificação WSDL que equivale à interface 'provides' de um componente de software. O Quadro 19.1 mostra parte da interface para um serviço simples que, dada uma data e um lugar específicos, como uma cidade dentro de um país, retorna as temperaturas máxima e mínima registradas naquele lugar e data. A mensagem de entrada também especifica se essas temperaturas deverão retornar em graus Celsius ou Fahrenheit.

No Quadro 19.1, a primeira parte da descrição mostra parte do elemento e o tipo de definição usado na especificação de serviço. Isso define os elementos PlaceAndDate, MaxMinTemp e InDataFault. Eu só inclui a especificação PlaceAndDate, que você pode pensar como um registro com três campos — cidade, país e data. Uma abordagem semelhante poderia ser usada para definir MaxMinTemp e InDataFault.

A segunda parte da descrição mostra como a interface de serviço é definida. Nesse exemplo, o serviço weatherInfo tem uma única operação, embora não existam restrições sobre o número de operações que podem ser definidas. A operação de weatherInfo tem um padrão *in-out* associado que significa que ele leva uma mensagem de entrada e gera uma de saída. A especificação WSDL 2.0 permite vários padrões de trocas de mensagens diferentes, como *in-out* e *out-only*, *in-optional-out*, *out-in* etc. As mensagens de entrada e saída, que se referem às definições feitas anteriormente na seção 'types', são definidas em seguida.

O grande problema com WSDL é que a definição de interface de serviço não inclui quaisquer informações sobre a semântica do serviço ou suas características não funcionais, como desempenho e confiança. É simplesmente uma descrição da assinatura de serviço (ou seja, as operações e seus parâmetros). O programador que planeja usar o serviço precisa definir o que o serviço realmente faz e o que significam os diferentes campos nas mensagens de entrada e saída. O desempenho e a confiança precisam ser descobertos por meio de experimentos. A documentação e os nomes significativos ajudam a compreender a funcionalidade oferecida, mas ainda é possível que os leitores não entendam o serviço.

Quadro 19.1 Parte de uma descrição WSDL para um *web service*

Defina alguns dos tipos usados. Suponha que o prefixo de *namespace* 'ws' se refira ao *namespace* URI para esquemas XML e o prefixo de *namespace* associado com essa definição seja *weathns*.

```
<types>
  <xs: schema targetNameSpace = "http://.../weathns"
      xmlns: weathns = "http://.../weathns" >
      <xs:element name = "PlaceAndDate" type = "pdrec" />
      <xs:element name = "MaxMinTemp" type = "mmtrec" />
      <xs:element name = "InDataFault" type = "errmess" />
      <xs:complexType name = "pdrec"
      <xs:sequence>
          <xs:element name = "cidade" type = "xs:string"/>
          <xs:element name = "país" type = "xs:string"/>
          <xs:element name = "dia" type = "xs:date"/>
        </xs:complexType>
        Definições de MaxMinType e InDataFault aqui
  </schema>
</types>
```

Agora, defina a interface e suas operações. Nesse caso, existe apenas uma operação para retornar as temperaturas máximas e mínimas.

```
<interface name = "weatherInfo" >
  <operation name = "getMaxMinTemps" pattern = "wsdlns: in-out">
  <input messageLabel = "In" element = "weathns: PlaceAndDate" />
  <output messageLabel = "Out" element = "weathns:MaxMinTemp" />
  <outfault messageLabel = "Out" element = "weathns:InDataFault" />
</operation>
</interface>
```

19.2 Engenharia de serviços

A engenharia de serviços é o processo de desenvolvimento de serviços para reúso em aplicações orientadas a serviços. Ela tem muito em comum com a engenharia de componentes. Os engenheiros de serviço precisam garantir que o serviço represente uma abstração reusável que poderia ser útil em diferentes sistemas. Geralmente, eles devem projetar e desenvolver uma funcionalidade útil associada com essa abstração e assegurar que o serviço seja robusto e confiável. Eles devem documentar o serviço para que possam ser descobertos e compreendidos pelos usuários em potencial.

Existem três estágios lógicos no processo de engenharia de serviço, conforme mostrados na Figura 19.5. São os seguintes:

1. Identificação de serviço candidato, em que você identifica os possíveis serviços que podem ser implementados e define os requisitos de serviço.
2. Projeto de serviço, em que você projeta a lógica e as interfaces de serviço WSDL.
3. Implementação e implantação de seviço, em que você implementa e testa os serviços tornando-os disponíveis para uso.

Figura 19.5 O processo de engenharia de serviço

Conforme discutido no Capítulo 16, o desenvolvimento de um componente reusável pode começar com um componente existente que já foi implementado e usado em uma aplicação. O mesmo é verdadeiro para serviços — o ponto de partida para esse processo será, muitas vezes, um serviço existente ou um componente que será convertido em um serviço. Nessa situação, o processo de projeto envolve a generalização de componentes existentes, de forma que são removidas características específicas da aplicação. A implementação significa adaptar o componente adicionando interfaces de serviço e implementando as generalizações necessárias.

19.2.1 Identificação de serviço candidato

A noção básica da computação orientada a serviços é que os serviços deveriam apoiar os processos de negócios. Como cada organização tem uma ampla gama de processos, existem muitos serviços possíveis, que podem ser implementados. A identificação de serviço candidato envolve, portanto, compreensão e análise dos processos de negócios da organização para decidir quais serviços reusáveis poderiam ser implementados para dar suporte a esses processos.

Erl sugere que existem três tipos fundamentais de serviços que podem ser identificados:

1. *Serviços utilitários.* São os que implementam alguma funcionalidade geral, que pode ser usada por diferentes processos de negócios. Um exemplo de serviço utilitário é um serviço de conversão de moeda que poderá ser acessado para calcular a conversão de uma moeda (por exemplo, dólares) para outra (por exemplo, euros).
2. *Serviços de negócio.* São os serviços que estão associados a uma função específica dos negócios. O exemplo de uma função de negócios em uma universidade seria o registro de alunos de um curso.
3. *Serviços de coordenação ou de processo.* São os serviços que suportam um processo de negócios amplo que geralmente envolve atividades e atores diferentes. Um exemplo de serviço de coordenação em uma empresa é um serviço de pedidos em que os pedidos são feitos, os produtos são aceitos e os pagamentos são efetuados.

Erl também sugere que serviços podem ser pensados como orientados a tarefas ou a entidades. Os serviços orientados a tarefas são aqueles associados com alguma atividade, e os orientados a entidades são como objetos. Eles são associados com uma entidade de negócios, como, por exemplo, um formulário de pedido de emprego. A Tabela 19.1 mostra alguns exemplos de serviços que são orientados a tarefas ou a entidades. Os serviços de negócio ou utilitários podem ser orientados a entidades ou a tarefas, mas os serviços de coordenação são sempre orientados a tarefas.

Seu objetivo na identificação de serviço candidato deve ser identificar os serviços que são logicamente coerentes, independentes e reusáveis. A classificação de Erl é útil nesse sentido, pois ele sugere como descobrir os serviços reusáveis, observando as entidades de negócios e as atividades de negócios. No entanto, identificar candidatos a serviços pode ser difícil porque é necessário considerar como os serviços serão usados. É preciso pensar em possíveis candidatos e, então, analisar uma série de questões sobre eles para ver se tendem a ser serviços úteis. As possíveis questões para identificar os serviços potencialmente reusáveis são:

1. Para um serviço orientado a entidades, esse serviço é associado com uma única entidade lógica usada em diferentes processos de negócios? Quais operações são normalmente executadas sobre essa entidade que devem ser apoiadas?
2. Para um serviço orientado a tarefas, é uma tarefa efetuada por pessoas diferentes na organização? As pessoas estarão dispostas a aceitar a padronização inevitável que ocorre quando um único suporte é prestado?
3. O serviço é independente (ou seja, até que ponto ele depende da disponibilidade de outros serviços)?

Tabela 19.1 A classificação de serviços

	Utilitário	Negócios	Coordenação
Tarefa	Conversor de moeda Localizador de funcionário	Validar formulário de reclamação Avaliar classificação de crédito	Processar despesas de reclamações Pagar fornecedor externo
Entidade	Verificador de estilo de documento Conversor de formulário Web para XML	Formulário de despesas Formulário de solicitação de estudante	

4. Para seu funcionamento, o serviço precisa manter o estado? Os serviços não têm estados, o que significa que eles não mantêm o estado interno. Se forem necessárias informações de estado, um banco de dados terá de ser usado e isso pode limitar a reusabilidade de serviço. Em geral, serviços em que o estado é passado para o serviço são mais fáceis de reusar, pois não é necessária nenhuma ligação de banco de dados.
5. O serviço poderia ser usado por clientes fora da organização? Por exemplo, um serviço orientado a entidades associado com um catálogo pode ser acessado por usuários internos e externos.
6. Os diferentes usuários do serviço podem ter diferentes requisitos não funcionais? Se a resposta for positiva, isso sugere que talvez seja aplicada mais de uma versão de um serviço.

As respostas a essas questões ajudam a selecionar e refinar abstrações que podem ser implementadas como serviços. No entanto, não existe uma forma única e regular para decidir quais são os melhores serviços e, portanto, a identificação de serviço é um processo baseado em experiência e habilidade.

A saída do processo de seleção de serviço é um conjunto de serviços identificados e requisitos associados para esses serviços. Os requisitos funcionais de serviços devem definir o que o serviço deve fazer. Os requisitos não funcionais devem definir requisitos de proteção, desempenho e disponibilidade do serviço.

Para ajudá-lo a compreender o processo de identificação e implementação de serviços candidatos, considere o seguinte exemplo:

> *Uma grande empresa, que vende equipamentos de informática, organizou preços especiais para configurações aprovadas para alguns clientes. Para facilitar a encomenda automatizada, a empresa pretende produzir um serviço de catálogo que permitirá aos clientes selecionar o equipamento de que precisam. Ao contrário de um catálogo de consumidor, os pedidos não serão feitos diretamente por meio de uma interface de catálogo. Em vez disso, os produtos serão ordenados por meio do sistema de aquisição, baseado na Web, de cada empresa que acessa o catálogo como um* web service. *A maioria das empresas tem seus próprios procedimentos de orçamentação e aprovação de pedidos e seu próprios processos de pedidos devem ser seguidos quando um pedido é feito.*

O serviço de catálogo é um exemplo de serviço orientado a entidades que oferece suporte a operações de negócios. Os requisitos de serviço de catálogos funcionais são:

1. Uma versão específica de catálogo é fornecida para cada empresa do usuário. Isso inclui as configurações e os equipamentos que podem ser ordenados por funcionários da empresa do cliente e os preços combinados para cada item.
2. O catálogo deve permitir a um funcionário do cliente fazer o download de uma versão do catálogo para navegação *off-line.*
3. O catálogo deve permitir que os usuários comparem as especificações e os preços de até seis itens do catálogo.
4. O catálogo deve fornecer aos usuários facilidades de navegação e pesquisa.
5. Os usuários do catálogo devem ser capazes de descobrir a data prevista para a entrega de um dado número de itens específicos do catálogo.
6. Os usuários do catálogo serão capazes de colocar 'ordens virtuais', em que os itens necessários serão reservados para eles por 48 horas. As ordens virtuais devem ser confirmadas por uma ordem real colocada por um sistema de aquisição. Ela deve ser recebida dentro de 48 horas a partir da ordem virtual.

Além desses requisitos funcionais, o catálogo tem alguns requisitos não funcionais:

1. O acesso ao serviço de catálogo deve ser restrito apenas a funcionários de organizações autorizadas.
2. Os preços e as configurações oferecidas para um cliente serão confidenciais e não estarão disponíveis para os funcionários de qualquer outro cliente.
3. O catálogo estará disponível sem interrupção do serviço das 7h às 11h.
4. O serviço de catálogo deve ser capaz de processar até dez solicitações por segundo em momentos de pico.

Observe que não existe um requisito não funcional relacionado ao tempo de resposta do serviço de catálogo. Isso depende do tamanho do catálogo e do número esperado de usuários simultâneos. Como esse não é um serviço de tempo crítico, não é necessário especificá-lo nesse estágio.

19.2.2 Projeto de interface de serviço

Uma vez que você selecionou serviços candidatos, o próximo estágio do processo de engenharia de serviço é projetar as interfaces de serviço. Isso envolve a definição das operações associadas com o serviço e seus parâmetros. Você também deve pensar cuidadosamente sobre o projeto de operações e mensagens de serviço. Seu objetivo deve ser minimizar o número de trocas de mensagens que deve acontecer para se completar a solicitação de serviço. É necessário garantir que tanta informação quanto possível seja passada para o serviço por uma mensagem, em vez de interações síncronas de serviço.

Você também deve se lembrar de que os serviços não têm estado, e gerenciar estado específico da aplicação do serviço é de responsabilidade do usuário do serviço, e não do serviço em si. Você pode, por conseguinte, ter de passar essas informações de estado para e dos serviços em mensagens de entrada e saída.

Existem três estágios de projeto de interface de serviço:

1. *Projeto lógico de interface*, em que você identifica as operações associadas ao serviço, suas entradas e saídas e as exceções associadas a essas operações.
2. *Projeto de mensagem*, em que você cria a estrutura das mensagens que são enviadas e recebidas pelo serviço.
3. *Desenvolvimento WSDL*, em que você traduz o projeto lógico e de mensagem para uma descrição da interface abstrata escrita em WSDL.

O primeiro estágio — projeto lógico de interface — começa com os requisitos de serviço e define os parâmetros e nomes de operação. Nesse estágio, você também deve definir as exceções que podem surgir quando uma operação de serviço é chamada. A Tabela 19.2 e a Figura 19.6 mostram as operações que implementam os requisitos e as entradas, saídas e exceções para cada uma das operações de catálogo. Nesse estágio, não é necessário especificar essas operações em detalhes — você adicionará os detalhes no próximo estágio do processo.

A definição de exceções e de como elas podem ser comunicadas aos usuários de serviços é particularmente importante. Os engenheiros de serviço não sabem como seus serviços serão usados. Geralmente, é imprudente fazer suposições de que os usuários de serviços compreenderão completamente a especificação do serviço. As mensagens de entrada podem estar incorretas, portanto, você deve definir exceções que reportam entradas incorretas para o cliente de serviço. Geralmente, é uma boa prática no desenvolvimento de componentes reusáveis deixar todos os tratamentos de exceções para o usuário do componente. O desenvolvedor de serviço não deve impor suas opiniões sobre como as exceções devem ser tratadas.

Depois de estabelecer uma descrição lógica informal do que o serviço deve fazer, o próximo estágio é definir a estrutura das mensagens de entrada e saída e os tipos usados nessas mensagens. XML é uma notação inadequada para se usar nesse estágio. Creio que é melhor representar as mensagens como objetos e defini-las usando a UML ou em uma linguagem de programação, tal como Java. Elas podem ser convertidas em XML manualmente ou automaticamente. A Tabela 19.3 mostra a estrutura de mensagens de entrada e saída para a operação getDelivery no serviço de catálogo.

Tabela 19.2 Descrições funcionais de operações de serviços de catálogo

Operação	Descrição
MakeCatalog	Cria uma versão do catálogo sob medida para um cliente específico. Inclui um parâmetro opcional para criar uma versão em PDF para download do catálogo.
Compare	Fornece uma comparação de até seis características (por exemplo, preço, dimensões, velocidade do processador etc.) de até quatro itens do catálogo.
Lookup	Exibe todos os dados associados a um item especificado do catálogo.
Search	Essa operação usa uma expressão lógica e procura o catálogo de acordo com essa expressão. Ela exibe uma lista de todos os itens que correspondam à expressão da busca.
CheckDelivery	Retorna a data de entrega prevista para um item ordenado naquele dia.
MakeVirtualOrder	Reserva o número de itens encomendado por um cliente e fornece informações sobre o item para o sistema de aquisição do cliente.

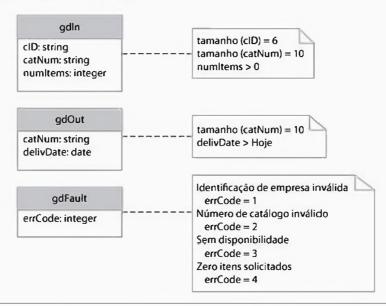
Figura 19.6 Projeto de interface de catálogo

Observe como adicionei detalhes na descrição pela anotação de diagrama UML com restrições. Elas definem o comprimento de *strings* que representam a empresa e o item de catálogo e especificam que o número de itens deve ser maior que zero e que a entrega deve ser após a data atual. As anotações também mostram quais códigos de erro são associados a cada possível defeito.

Tabela 19.3 Definição em UML de mensagens de entrada e saída

Operação	Entradas	Saídas	Exceções
MakeCatalog	*mcIn* Id de empresa PDF-bandeira	*mcOut* URL do catálogo para essa empresa	*mcFault* Id de empresa inválida
Compare	*compIn* Id de empresa Atributo de entrada (até 6) Número de catálogo (até 4)	*compOut* URL da página apresentando tabela de comparação	*compFault* Id de empresa inválida Atributo desconhecido Número de catálogo inválido
Lookup	*lookIn* Id de empresa Número de catálogo	*lookOut* URL da página com as informações de item	*lookFault* Id de empresa inválida Número de catálogo inválido
Search	*searchIn* Id de empresa *String* de busca	*searchOut* URL da página Web com resultados de busca	*searchFault* Id de empresa inválida *String* de busca mal formado
CheckDelivery	*gdIn* Id de empresa Número de catálogo Número de itens necessários	*gdOut* Número de catálogo Data esperada de espera	*gdFault* Id de empresa inválida Número de catálogo inválido Sem disponibilidade Zero item solicitado
PlaceOrder	*poIn* Id de empresa Número de itens necessários Número de catálogo	*poOut* Número de catálogo Número de itens requisitados Data prevista de entrega Estimativa de preço unitário Estimativa de preço total	*poFault* Id de empresa inválido Número de catálogo inválido Zero item requisitado

O estágio final do processo de projeto de serviço é a tradução do projeto de interface de serviço em WSDL. Conforme discutido na seção anterior, uma representação em WSDL é longa e detalhada, portanto, é fácil cometer erros nesse estágio se você fizer isso manualmente. No entanto, a maioria dos ambientes de programação que dão suporte ao desenvolvimento orientado a serviços (por exemplo, o ambiente ECLIPSE) inclui ferramentas que podem traduzir uma descrição lógica de interface em sua representação em WSDL correspondente.

19.2.3 Implementação e implantação de serviço

Depois de ter identificado os serviços candidatos e projetado suas interfaces, o estágio final do processo de engenharia de serviço é a implementação de serviço. Essa implementação pode envolver programação de serviço usando uma linguagem de programação padrão, como Java ou C#. Essas linguagens incluem bibliotecas com amplo suporte para o desenvolvimento de serviços.

Como alternativa, os serviços podem ser desenvolvidos pela implementação de interfaces de serviço para os componentes existentes ou, como discuto adiante, para sistemas legados. Isso significa que os ativos de software que já provaram ser úteis podem ser amplamente disponibilizados. No caso de sistemas legados, pode significar que a funcionalidade do sistema pode ser acessada por novas aplicações. Você também pode desenvolver novos serviços, definindo composições de serviços existentes. Eu discuto essa abordagem para o desenvolvimento de serviços na Seção 19.3.

Depois que um serviço foi implementado, ele precisa ser testado antes da implantação. Trata-se de examinar e dividir as entradas de serviço (como explicado no Capítulo 8), criando-se mensagens de entrada que refletem essas combinações de entradas, e, em seguida, verificar que as saídas são esperadas. Você sempre deve tentar gerar exceções durante o teste para verificar se o serviço pode lidar com entradas inválidas. Estão disponíveis ferramentas de teste que permitem que os serviços sejam examinados e testados e que geram testes a partir de uma especificação em WSDL. No entanto, elas só podem testar a conformidade da interface de serviço para o WSDL, e não o comportamento funcional do serviço.

A implantação do serviço, estágio final do processo, envolve disponibilizar o serviço para uso em um servidor Web. A maioria dos softwares de servidor torna esse processo muito simples. É necessário apenas instalar o arquivo que contém o serviço executável em um diretório específico. Em seguida, ele fica disponível para uso, automaticamente. Se o serviço deve ser disponibilizado publicamente, você deve fornecer informações para os usuários externos do serviço. Essas informações ajudam os potenciais usuários externos a decidir se o serviço é suscetível de satisfazer a suas necessidades e se eles podem confiar em você, como um provedor de serviços, para oferecer o serviço de maneira confiável e segura. As informações que você pode incluir em uma descrição de serviço podem ser:

1. Informações sobre seu negócio, contatos etc. Isso é importante por razões de confiança. Os usuários de um serviço precisam ter certeza de que ele não se comportará maliciosamente. As informações sobre o provedor de serviços permitem verificação de suas credenciais com as agências de informação de negócios.
2. Uma descrição informal da funcionalidade fornecida pelo serviço. Isso ajuda os potenciais usuários a decidir se o serviço é o que eles querem. No entanto, a descrição funcional é em linguagem natural, portanto, não é uma descrição semântica inequívoca do que faz o serviço.
3. Uma descrição detalhada dos tipos e semântica de interface.
4. Informações de assinatura que permitem aos usuários se registrarem para obter informações sobre atualizações do serviço.

Como já discutido, um problema geral com as especificações de serviço é que seu comportamento funcional geralmente é especificado informalmente, como uma descrição em linguagem natural. As descrições em linguagem natural são fáceis de serem lidas, mas estão sujeitas a interpretações incorretas. Para resolver esse problema, existe uma comunidade ativa de pesquisa, interessada em investigar como a semântica dos serviços pode ser especificada. A abordagem mais promissora para especificação de semântica é baseada em uma descrição que se relaciona com a ontologia, em que o significado específico de termos em uma descrição é definido em uma ontologia. As ontologias são uma forma de padronizar as formas como a terminologia é usada e definir as relações entre diferentes termos. Elas têm sido cada vez mais usadas para ajudar a atribuir semântica para descrições em linguagem natural. Uma linguagem chamada OWL-S foi desenvolvida para descrever ontologias de *web services* (OWL_Services_Coalition, 2003).

19.2.4 Serviços de sistemas legados

Sistemas legados são antigos sistemas de software usados por uma organização. Geralmente, eles contam com tecnologia obsoleta, mas ainda são essenciais para os negócios. Pode não ser efetivo reescrever ou substituir esses sistemas, e muitas organizações gostariam de usá-los em conjunto com os sistemas mais modernos. Um dos usos mais importantes de serviços é implementar 'empacotadores' para sistemas legados que fornecem acesso a funções e dados de um sistema. Esses sistemas podem ser acessados pela Web e integrados com outras aplicações.

Para ilustrar isso, imagine que uma grande empresa mantém um inventário de seus equipamento em um banco de dados associado que acompanha os reparos e a manutanção de equipamentos. Isso mantém o acompanhamento de solicitações de manutenção que foram realizadas para diferentes partes do equipamento, que tipo de manutenção regular está programada, quando as manutenções foram realizadas, quanto tempo foi gasto em manutenção etc. Esse sistema legado foi originalmente usado para gerar listas de trabalho diariamente, para o pessoal de manutenção, mas, ao longo do tempo, novos recursos foram adicionados. Esses recursos fornecem dados sobre quanto foi gasto em manutenção com cada peça de equipamento e informações para ajudar a avaliar o custo de manutenção a ser realizada por prestadores de serviços externos. O sistema é executado como um sistema cliente-servidor com software especial para cliente executando em um PC.

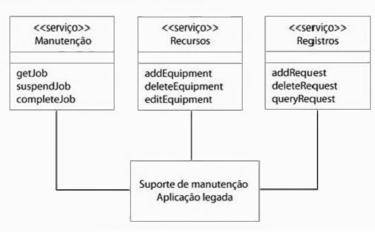
A empresa agora pretende fornecer acesso em tempo real para esse sistema a partir de terminais portáteis usados pelo pessoal de manutenção. Eles atualizarão o sistema com o tempo e os recursos gastos em manutenção e consultarão o sistema para encontrar seu próximo trabalho de manutenção. Além disso, a equipe de *call center* requer acesso ao sistema para registar pedidos de manutenção e verificar seus *status*.

É praticamente impossível melhorar o sistema para apoiar esses requisitos. Então, a empresa decide fornecer novas aplicações para a manutenção e para a equipe de *call center*. Essas aplicações contam com o sistema legado, que deve ser usado como uma base para implementar uma série de serviços. Isso é ilustrado na Figura 19.7, em que eu usei um estereótipo UML para indicar um serviço. Novas aplicações trocam mensagens com esses serviços para acessar a funcionalidade do sistema legado.

Alguns dos serviços fornecidos são os seguintes:

1. *Um serviço de manutenção*. Inclui operações para recuperar um trabalho de manutenção de acordo com seu número de tarefa, prioridade e localização geográfica, e carrega os detalhes da manutenção feita no banco de dados de manutenção. O serviço também fornece operações que permitem que um trabalho de manutenção que foi iniciado, mas está incompleto, seja suspenso e reiniciado.
2. *Um serviço de recursos*. Inclui operações para adicionar e excluir novos equipamentos e modificar as informações associadas com os equipamentos no banco de dados.
3. *Um serviço de registros*. Inclui operações para adicionar uma nova solicitação de serviço, eliminar as solicitações de manutenção e consultar o *status* das solicitações pendentes.

Observe que o sistema legado existente não é representado como um único serviço. Em vez disso, os serviços que são desenvolvidos para acessar o sistema legado são coerentes e apoiam uma única área de funcionalidade. Isso reduz sua complexidade e facilita a compreensão e o reúso em outras aplicações.

Figura 19.7 Serviços fornecendo acesso a um sistema legado

19.3 Desenvolvimento de software com serviços

O desenvolvimento de software usando serviços baseia-se na ideia de compor e configurar serviços para criar novos serviços compostos. Estes podem ser integrados com uma interface de usuário implementada em um *browser* para criar uma aplicação Web ou podem ser usados como componentes em alguma outra composição de serviço. Os serviços envolvidos na composição podem ser especialmente desenvolvidos para a aplicação, podem ser desenvolvidos dentro de uma empresa de serviços de negócios ou podem ser serviços de um fornecedor externo.

Atualmente, muitas empresas estão convertendo suas aplicações corporativas em sistemas orientados a serviços, em que o bloco básico de construção de aplicações é um serviço, e não um componente. Isso abre a possibilidade de reúso mais generalizado dentro da empresa. O próximo estágio será o desenvolvimento de aplicações interorganizacionais entre fornecedores confiáveis, que usarão serviços uns dos outros. A realização final da visão de longo prazo de SOA contará com o desenvolvimento de um 'mercado de serviços', em que os serviços serão comprados de fornecedores externos.

A composição de serviços pode ser usada para integrar os processos de negócios separados para fornecer um processo integrado oferecendo funcionalidade mais ampla, como por exemplo uma companhia aérea que pretende fornecer um pacote de férias completo para os viajantes. Assim como reservar seus voos, os viajantes podem também reservar hotéis no local de sua preferência, organizar o aluguel de carros ou reservar um táxi no aeroporto, procurar um guia de viagens e fazer reservas para visitar as atrações locais. Para criar essa aplicação, a companhia aérea compõe seu próprio serviço de reserva com os serviços oferecidos por uma agência de reservas de hotéis, aluguel de carro e companhias de táxi e serviços de reserva oferecidos pelos proprietários de atrações locais. O resultado final é um serviço único que integra os serviços de diferentes provedores.

Você pode pensar nesse processo como uma sequência de etapas separadas, como mostra a Figura 19.8. As informações são passadas de uma etapa para a próxima — por exemplo, a empresa de aluguel de carros é informada do horário de chegada do voo. A sequência de etapas é chamada *workflow* — um conjunto de atividades ordenadas no tempo, com cada atividade efetuando alguma parte do trabalho. Um *workflow* é um modelo de processo de negócios (ou seja, estabelece as etapas envolvidas em alcançar um objetivo particular que é importante para uma empresa). Nesse caso, o processo de negócios é o serviço de reservas de férias oferecido pela companhia aérea.

O *workflow* é uma ideia simples, e o cenário de fazer reservas para as férias parece ser simples. Na prática, a composição de serviço é muito mais complexa do que esse modelo simples representa. Por exemplo, você deve considerar a possibilidade de falha de serviço e incluir mecanismos para lidar com essas falhas. Você também precisa ter em mente as possíveis demandas excepcionais realizadas por usuários da aplicação. Digamos que um viajante se machucou e uma cadeira de rodas precisa ser alugada e entregue no aeroporto. A implementação e composição disso exigiria serviços extras, além de etapas adicionais a serem adicionadas ao *workflow*.

Você deve ser capaz de lidar com situações em que o *workflow* precisa ser alterado porque a execução normal de um dos serviços geralmente resulta em uma incompatibilidade com alguma outra execução de serviço. Por exemplo, digamos que um voo é reservado para sair no dia 1º de junho e retornar no dia 7 de junho. Em seguida, o *workflow* passa para a fase de reserva de hotel. No entanto, o *resort* está hospedando uma grande convenção até o dia 2 de junho, portanto não há quartos disponíveis. O serviço de reserva de hotel relata essa falta de disponibilidade. Não se trata de uma falha. A falta de disponibilidade é uma situação comum. Portanto, em seguida, você precisa 'desfazer' a reserva do voo e passar as informações sobre a falta de disponibilidade para o usuário, que precisará decidir se deseja alterar as datas ou o *resort*. Na terminologia do *workflow*, isso é chamado 'ação de compensação'. Ações de compensação são usadas para desfazer ações que já foram concluídas, mas que devem ser alteradas como resultado de atividades posteriores do *workflow*.

Figura 19.8 *Workflow* do pacote de férias

O processo de projeto de novos serviços reusando serviços existentes é essencialmente um processo de projeto de software com reúso (Figura 19.9). O projeto com reúso envolve, inevitavelmente, compromissos de requisitos. Os requisitos 'ideais' para o sistema precisam ser modificados para refletir os serviços disponíveis, cujos custos são incluídos no orçamento e cuja qualidade do serviço é aceitável.

Na Figura 19.9, eu mostro seis estágios essenciais do processo de construção de serviços por composição:

1. *Formular* workflow *preliminar*. Nesse estágio inicial de projeto de serviço, você usa os requisitos para o serviço composto como uma base para a criação de um projeto de serviço 'ideal'. Nesse estágio, você deve criar um projeto bastante abstrato com a intenção de adicionar detalhes depois de saber mais sobre os serviços disponíveis.
2. *Descobrir serviços*. Durante esse estágio do processo, você pesquisa por registros de serviço ou catálogos para descobrir quais serviços existem, quem fornece esses serviços e os detalhes de fornecimento de serviços.
3. *Selecionar possíveis serviços*. Do conjunto de serviços candidatos que você descobriu, selecionam-se serviços que possam implementar as atividades de *workflow*. Os critérios de seleção obviamente incluirão a funcionalidade dos serviços oferecidos, bem como o custo e a qualidade destes (capacidade de resposta, disponibilidade etc.). Você pode decidir escolher um número de serviços funcionalmente equivalentes, que podem ser ligados a uma atividade de *workflow* dependendo dos detalhes dos custos e da qualidade de serviço.
4. *Refinar* workflow. Com base nas informações sobre os serviços que foram selecionados, você pode refinar o *workflow*. Isso envolve adicionar detalhes para a descrição abstrata e, talvez, adicionar ou remover atividades de *workflow*. Em seguida, você pode repetir os estágios de descoberta e seleção de serviços. Uma vez que um conjunto de serviços tenha sido escolhido e o projeto de *workflow* final esteja estabelecido, você passa para o próximo estágio do processo.
5. *Criar programa de* workflow. Durante esse estágio, o projeto abstrato de *workflow* é transformado em um programa executável, e a interface de serviço é definida. Pode-se usar uma linguagem de programação convencional, tais como Java ou C#, para implementação de serviço, ou uma linguagem de *workflow*, como WS-BPEL. Conforme discutido na seção anterior, a especificação de interface de serviço deve ser escrita em WSDL. Esse estágio também pode envolver criações de interfaces de usuário baseadas em Web para permitir que o novo serviço seja acessado a partir de um *browser* de Web.
6. *Testar serviço ou aplicação completa*. O processo de teste de serviço de composição completo é mais complexo do que o teste de componente em situações de uso de serviços externos. Na Seção 19.3.2, eu discuto os problemas de teste.

No restante deste capítulo, foca-se no projeto e testes de *workflow*. Na prática, a descoberta de serviços não parece ser um grande problema. A maioria dos reúsos de serviços ainda ocorre dentro das organizações, onde os serviços podem ser descobertos a partir de registros internos e comunicações informais entre os engenheiros de software. Os mecanismos padrões de busca podem ser usados para descobrir serviços acessíveis ao público.

19.3.1 Implementação e design de *workflow*

O projeto de *workflow* envolve a análise de processos de negócios existentes ou planejados para compreender as diferentes atividades realizadas e como as informações são trocadas. Em seguida, você define o novo processo de negócios em uma notação de projeto de *workflow*, definindo os estágios envolvidos na adoção do processo e as informações que passam entre os diferentes estágios do processo. No entanto, os processos existentes podem

Figura 19.9 Construção de serviço por composição

ser informais e dependentes das habilidades e capacidades das pessoas envolvidas —pode não haver uma maneira 'normal' de trabalho ou definição de processo. Em tais casos, você precisa usar seu conhecimento do processo atual para criar um *workflow* que atinja os mesmos objetivos.

Os *workflows* representam modelos de processo de negócios e são geralmente representados por meio de uma notação gráfica, como diagramas de atividades da UML ou BPMN, notação de modelagem de processo de negócio (WHITE, 2004a; WHITE e MIERS, 2008). Estes oferecem características semelhantes (WHITE, 2004b). É provável que BPMN e diagramas de atividades UML serão integrados no futuro, e um padrão para modelagem de *workflow* definido será baseado nessa linguagem integrada. Para os exemplos neste capítulo, eu uso BPMN.

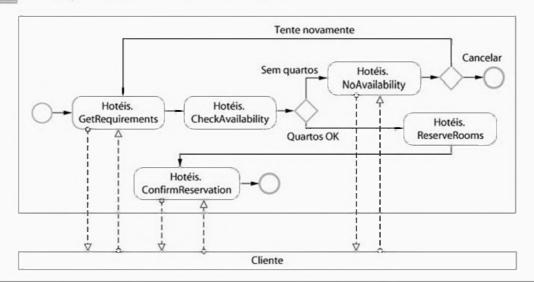
BPMN é uma linguagem gráfica razoavelmente fácil de entender. Os mapeamentos foram definidos para traduzir a linguagem para as descrições baseadas em XML, de baixo nível, em WS-BPEL. Consequentemente, BPMN está de acordo com os padrões de *web service* mostrados na Figura 19.2.

A Figura 19.10 é um exemplo de um modelo simples em BPMN de parte do cenário do pacote de férias anteriormente mencionado. O modelo mostra um *workflow* simplificado para a reserva de hotel e pressupõe a existência de um serviço de hotéis com operações associadas chamadas GetRequirements, CheckAvailability, ReserveRooms, NoAvailability, ConfirmReservation e CancelReservation. O processo envolve a obtenção de requisitos do cliente, verificação de disponibilidade de quartos e, em seguida, se há disponibilidade, uma reserva nas datas requeridas.

Esse modelo apresenta alguns dos principais conceitos de BPMN, que são usados para criar modelos de *workflow*:

1. As atividades são representadas por um retângulo com cantos arredondados; uma atividade pode ser executada por seres humanos ou por serviços automatizados.
2. Os eventos são representados por círculos; um evento é algo que acontece durante um processo de negócios. Um círculo simples é usado para representar um evento que se inicia, e um círculo mais escuro, para representar um fim de um evento. Um círculo duplo (não mostrado) é usado para representar um evento intermediário. Os eventos podem ser eventos de relógio, permitindo que o *workflow* seja executado periodicamente ou com horário definido.
3. Um diamante representa um *gateway*. Um *gateway* é um estágio do processo em que algumas escolhas são feitas. A Figura 19.10 mostra o exemplo de uma escolha feita a respeito da disponibilidade de quartos.
4. Uma seta sólida é usada para mostrar a sequência de atividades. Uma seta tracejada representa o fluxo de mensagens entre as atividades. Na Figura 19.10, essas mensagens são passadas entre o serviço de reservas de hotel e o cliente.

Essas características principais são suficientes para descrever a essência da maioria dos *workflows*. No entanto, BPMN inclui muitos recursos adicionais, que não tenho espaço para descrever aqui. Eles adicionam informações para a descrição de processo de negócio que lhe permite ser traduzido automaticamente para um serviço exe-

Figura 19.10 Um fragmento de um *workflow* de reserva de hotel

cutável. Portanto, *web services*, baseados em composições de serviços descritos em BPMN, podem ser gerados diretamente a partir de um modelo de processo de negócios.

A Figura 19.10 mostra o processo definido em uma organização, a empresa que fornece um serviço de reservas. No entanto, o principal benefício de uma abordagem orientada a serviços é que ela oferece suporte à computação interorganizacional, o que significa que um processamento envolve serviços em diferentes empresas. Isso é representado em BPMN através do desenvolvimento de *worflows* separados para cada uma das organizações envolvidas com as interações entre eles.

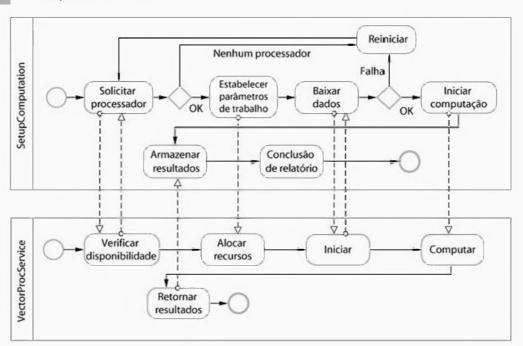
Para ilustrar isso, há um exemplo diferente, retirado de computação de alto desempenho. Propôs-se uma abordagem orientada a serviços para permitir que recursos como computadores de alto desempenho sejam compartilhados. Nesse exemplo, suponha que um computador de processamento de vetores (uma máquina capaz de efetuar processamentos paralelos em vetores de valores) seja oferecido como um serviço (**VectorProcService**) por um laboratório de pesquisa. Ele é acessado por meio de um serviço chamado **SetupComputation**. Esses serviços e suas interações são mostrados na Figura 19.11.

Nesse exemplo, o *workflow* para o serviço SetupComputation solicita acesso a um processador de vetores e, se um processador está disponível, estabelece o processamento requerido e baixa dados para o serviço de processamento. Uma vez que a computação é concluída, os resultados são armazenados no computador local. O *workflow* para **VectorProcService** verifica se um processador disponível aloca recursos para a computação, inicia o sistema, realiza o processamento e retorna os resultados para o serviço de cliente.

Em termos de BPMN, o *workflow* para cada organização é representado em um *pool* separado. Ele é mostrado graficamente, fechando o *workflow*, para cada participante do processo, em um retângulo, com o nome escrito na vertical, na borda esquerda. Os *workflows* definidos em cada *pool* são coordenados por troca de mensagens. O fluxo de sequência entre as atividades em diferentes *pools* não é permitido. Em situações nas quais diferentes partes de uma organização estejam envolvidas em um *workflow*, isso pode ser mostrado separando-se *pools* em 'raias' nomeadas. Cada raia mostra parte das atividades da organização.

Uma vez que tenha sido projetado um modelo de processo de negócios, este precisa ser refinado, dependendo dos serviços que foram descobertos. A Figura 19.9 mostra que o modelo pode passar por um número de iterações até que um projeto que permita um máximo de reúso de serviços disponíveis seja criado.

Assim que o projeto final estiver disponível, ele deve ser convertido em um programa executável. Isso pode envolver duas atividades:

Figura 19.11 Interação de *workflows*

1. Implementação dos serviços que não estão disponíveis para reúso de execução. Como os serviços são independentes de linguagens de implementação, eles podem ser escritos em qualquer linguagem. Os ambientes de desenvolvimento Java e C# oferecem suporte para a composição de *web services*.
2. Geração de uma versão executável do modelo de *workflow*. Normalmente, isso envolve a tradução do modelo em WS-BPEL, automática ou manualmente. Embora existam várias ferramentas disponíveis para automatizar o processo de BPMN-WS-BPEL, existem algumas circunstâncias em que é difícil gerar o código em WS-BPEL legível de um modelo *workflow*.

Para fornecer suporte direto para a implementação de composição de *web services*, foram desenvolvidos vários padrões de *web services*. Como expliquei na introdução do capítulo, a linguagem-padrão baseada em XML é a WS-BPEL (*Business Process Execution Language*), que é uma 'linguagem de programação' para controlar as interações entre os serviços. Isso é suportado por padrões complementares, como WS-Coordination (CABRERA et al., 2005), que são usados para especificar como os serviços são coordenados, e WS-CDL (*Choreography Description Language*) (KAVANTZAS et al., 2004), que é um meio para definir as trocas de mensagens entre os participantes (ANDREWS et al., 2003).

19.3.2 Teste de serviços

O teste é importante em todos os processos de desenvolvimento de sistema, pois ele demonstra se o sistema satisfaz a seus requisitos funcionais e não funcionais e detecta os defeitos ocorridos durante o processo de desenvolvimento. Muitas técnicas de testes, como inspeções de programa e testes de cobertura, dependem da análise do código-fonte do software. No entanto, quando os serviços são oferecidos por um fornecedor externo, o código-fonte de implementação do serviço não está disponível. Portanto, para o teste de sistema baseado em serviços não é possível usar técnicas comprovadas baseadas em códigos.

Assim como os problemas de entendimento de implementação de serviços, os testadores também podem enfrentar mais dificuldades ao testar serviços e composições de serviços:

1. Serviços externos estão sob o controle do fornecedor de serviço, em vez do usuário de serviço. O fornecedor de serviço pode retirar esses serviços a qualquer momento ou pode alterá-los, o que invalida quaisquer testes de aplicações anteriores. Esses problemas são tratados em componentes de software por meio da manutenção de versões diferentes de componente. Atualmente, no entanto, não há padrões propostos para lidar com versões de serviços.
2. A visão a longo prazo de SOA é que os serviços devem ser vinculados dinamicamente a aplicações orientadas a serviços. Isso significa que uma aplicação talvez não use sempre o mesmo serviço cada vez que for executada. Portanto, os testes podem ser bem-sucedidos quando uma aplicação é ligada a um determinado serviço, mas não se pode garantir que esse serviço será usado durante a execução real do sistema.
3. O comportamento não funcional de um serviço não depende simplesmente de como ele é usado pela aplicação que está sendo testada. Um serviço pode executar bem durante o teste, pois ele não está operando sob uma carga pesada. Na prática, o comportamento observado de serviço pode ser diferente por causa das demandas feitas por outros usuários dele.
4. O modelo de pagamento por serviços poderia tornar muito caros os testes de serviço. Existem diferentes modelos possíveis de pagamento — alguns serviços podem ser gratuitos, alguns podem ser pagos por assinatura, e outros, pagos de acordo com o uso. Se os serviços forem gratuitos, o provedor de serviços não desejará que sejam carregados por aplicações em testes; se a assinatura for necessária, um possível usuário do serviço poderá ficar relutante em assinar um contrato de assinatura antes de testar o serviço. Da mesma forma, se o uso for baseado em pagamento por uso, os usuários do serviço podem considerar os custos muito altos.
5. Eu já discuti a noção de ações de compensação, chamadas quando ocorre uma exceção e os compromissos feitos anteriormente (como uma reserva de voo) precisam ser revogados. Existe um problema no teste de tais ações, pois eles podem depender da falha de outros serviços. Assegurar que esses serviços realmente falhem durante o processo de teste pode ser muito difícil.

Esses problemas são particularmente agudos quando serviços externos são usados. Eles são menos graves quando os serviços são usados dentro da mesma empresa ou onde as empresas colaboram e confiam nos serviços oferecidos por seus parceiros. Nesses casos, o código-fonte pode estar disponível para orientar o processo de teste, e é pouco provável que o pagamento pelos serviços seja um problema. Resolver esses problemas de testes e produzir diretrizes, ferramentas e técnicas para testar as aplicações orientadas a serviços continua a ser uma importante questão de pesquisas.

PONTOS IMPORTANTES

- Arquitetura orientada a serviços é uma abordagem de engenharia de software em que os serviços reusáveis padronizados são os blocos básicos de construção para os sistemas de aplicação.
- Interfaces de serviço podem ser definidas em uma linguagem baseada em XML chamada WSDL. Uma especificação WSDL inclui uma definição dos tipos e operações de interface, o protocolo de ligação usado pelo serviço e a locação de serviço.
- Os serviços podem ser classificados como serviços utilitários que fornecem uma funcionalidade de uso geral, serviços de negócios que implementam parte de um processo de negócios ou serviços de coordenação que executam outros serviços.
- O processo de engenharia de serviço envolve a identificação de serviços candidatos à implementação, à definição de interface e à implementação de serviço e teste e à implantação de serviço.
- As interfaces de serviço podem ser definidas para sistemas legados que continuam úteis para uma organização. Em seguida, a funcionalidade do sistema legado pode ser reusada em outras aplicações.
- O desenvolvimento de software usando serviços baseia-se na ideia de que os programas são criados por composição e configuração de serviços para criar novos serviços compostos.
- Modelos de processos de negócios definem as atividades e o intercâmbio de informações que se realizam em um processo de negócios. Atividades em um processo de negócios podem ser implementadas por serviços para que o modelo de processo represente uma composição de serviço.

LEITURA COMPLEMENTAR

Existe uma imensa quantidade de material tutorial na Web discutindo todos os aspectos de *web services*. No entanto, acho os seguintes livros de Thomas Erl a melhor visão geral e descrição de serviços e padrões de serviço. Ao contrário da maioria dos livros, Erl inclui alguma discussão sobre questões de engenharia de software em computação orientada a serviços. Ele também escreveu livros mais especializados sobre o projeto de serviços e padrões de projeto SOA, embora esses sejam geralmente destinados a leitores com experiência de implementação de SOA.

Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services. O foco principal desse livro são as tecnologias subjacentes baseadas em XML (SOAP, WSDL, BPEL etc.), que são um *framework* para SOA. (ERL, T. *Service-Oriented Architecture*: A Field Guide to Integrating XML and Web Services. Prentice Hall, 2004.)

Service-Oriented Architecture: Concepts, Technology and Design. Esse é um livro mais geral sobre a engenharia de sistemas orientada a serviços. Existe um pouco de sobreposição com o texto anterior, mas Erl concentra-se principalmente em discutir como uma abordagem orientada a serviços pode ser usada em todos os estágios do processo de software. (ERL, T. *Service-Oriented Architecture*: Concepts, Technology and Design. Prentice Hall, 2005.)

'SOA realization: service design principles'. Esse curto artigo é um excelente panorama sobre as questões a serem consideradas na criação de serviços. (ARTUS, D. J. S. *IBM*, 2006.) Disponível em: <http://www.ibm.com/developerworks/webservices/library/ws-soa-design/>.

EXERCÍCIOS

19.1 Quais são as principais distinções entre serviços e componentes de software?

19.2 Explique por que as SOA devem basear-se em normas.

19.3 Usando a mesma notação, estenda o Quadro 19.1 para incluir as definições para MaxMinType e InDataFault. As temperaturas devem ser representadas como inteiros com um campo adicional que indica se a temperatura está em graus Fahrenheit ou graus Celsius. InDataFault deve ser um tipo simples que consiste em um código de erro.

19.4 Defina uma especificação de interface para os serviços de Avaliar classificação de crédito e Conversor de moeda mostrados na Tabela 19.1.

19.5 Projete mensagens de entrada e saída possíveis para os serviços mostrados na Figura 19.7. Você pode especificá-las em UML ou em XML.

19.6 Justificando sua resposta, dê sugestões de dois importantes tipos de aplicações em que você não recomenda o uso da arquitetura orientada a serviços.

19.7 Na Seção 19.2.1, apresentei o exemplo de uma empresa que desenvolveu um serviço de catálogo usado por sistemas de aquisição baseados na Web dos clientes. Usando BPMN, projete um *workflow* que usa esse serviço de catálogo para procurar e fazer pedidos para equipamentos de informática.

19.8 Explique o significado de uma 'ação de compensação' e, usando um exemplo, mostre por que essas ações podem precisar ser incluídas em *workflows*.

19.9 Para o exemplo do serviço de reserva de pacote de férias, projete um *workflow* que reservará o transporte terrestre para um grupo de passageiros que chegam a um aeroporto. Eles devem ter a possibilidade de reservar um táxi ou alugar um carro. Você pode assumir que as empresas de táxi e carro oferecem *web services* para fazer a reserva.

19.10 Usando um exemplo, explique detalhadamente por que é difícil realizar um teste completo de serviços que inclua ações de compensação.

REFERÊNCIAS

ANDREWS, T.; CURBERA, F.; GOLAND, Y.; KLEIN, J.; AL, E. Business Process Execution Language for Web Services. 2003. Disponível em: <http://www-128.ibm.com/developerworks/library/ws-bpel>.

CABRERA, L. F.; COPELAND, G.; AL, E. Web Services Coordination (WS-Coordination). 2005. Disponível em: <ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf>.

CARR, N. *The Big Switch*: Rewiring the World from Edison to Google, Reprint edition. Nova York: W.W. Norton & Co., 2009.

ERL, T. *Service-Oriented Architecture*: Concepts, Technology and Design. Upper Saddle River, NJ: Prentice Hall, 2005.

KAVANTZAS, N.; BURDETT, D.; RITZINGER, G. Web Services Choreography Description Language Version 1.0. 2004. Disponível em: <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>.

LOVELOCK, C.; VANDERMERWE, S.; LEWIS, B. *Services Marketing*. Englewood Cliffs, NJ: Prentice Hall, 1996.

NEWCOMER, E.; LOMOW, G. *Understanding SOA with Web Services*. Boston: Addison-Wesley, 2005.

Owl_Services_Coalition. OWL-S: Semantic Markup for Web Services. 2003. Disponível em: <http://www.daml.org/services/owl-s/1.0/owl-s.pdf>.

PAUTASSO, C.; ZIMMERMANN, O.; LEYMANN, F. RESTful Web Services *vs* "Big" Web Services: Making the Right Architectural Decision. *Proc. WWW 2008*, Beijing, China: 805-814, 2008.

RICHARDSON, L.; RUBY, S. *RESTful Web Services*. Sebastopol, Calif.: O'Reilly Media Inc., 2007.

TURNER, M.; BUDGEN, D.; BRERETON, P. Turning Software into a Service. *IEEE Computer*, v. 36, n. 10, 2003, p. 38-45.

WHITE, S. A. An Introduction to BPMN. 2004a. Disponível em: <http://www.bpmn.org/Documents/Introduction%20to%20BPMN>.

WHITE, S. A. Process Modelling Notations and Workflow Patterns. In: Fischer, L. (org.) *Workflow Handbook*. Lighthouse Point, Fla.: Future strategies Inc., 265-294, 2004.

WHITE, S. A.; MIERS, D. *BPMN Modeling and Reference Guide*: Understanding and Using BPMN. Lighthouse Point, Fla.: Future Strategies Inc., 2008.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 **20** 21 22 23 24 25 26

Software embutido

Objetivos

O objetivo deste capítulo é apresentar algumas das características de sistemas embutidos de tempo real e da engenharia de software de tempo real. Com a leitura deste capítulo, você:

- entenderá o conceito de software embutido, usado para controlar sistemas que devem reagir a eventos externos em seu ambiente;
- conhecerá o processo de projeto para sistemas de tempo real, em que os sistemas de software são organizados como um conjunto de processos colaborativos;
- compreenderá três padrões de arquitetura comumente usados no projeto de sistemas embutidos de tempo real;
- entenderá a organização dos sistemas operacionais de tempo real e o papel que eles desempenham em um sistema embutido de tempo real.

Conteúdo

Os computadores são usados para controlar uma vasta gama de sistemas, desde máquinas domésticas simples, controladores de jogos, até plantas inteiras de fabricação. Esses computadores interagem diretamente com dispositivos de hardware. Seu software deve reagir a eventos gerados pelo hardware e, muitas vezes, emitir sinais de controle em resposta a tais eventos. Esses sinais resultam em uma ação, como o início de uma chamada de telefone, o movimento de um caractere na tela, a abertura de uma válvula, ou a exibição de *status* do sistema. O software é embutido no hardware do sistema, muitas vezes em memória do tipo apenas leitura, e geralmente responde em tempo real a eventos no ambiente do sistema. Diz-se que o sistema de software tem um *deadline* para responder a eventos externos em tempo real e, se esse *deadline* for perdido, o sistema de hardware/software global não funcionará corretamente.

O software embutido é muito importante economicamente porque quase todos os dispositivos elétricos incluem software. Portanto, existem muitos sistemas de software embutido, mais do que outros tipos de sistema de software. Se você olhar em sua casa, poderá notar que existem três ou quatro computadores pessoais, mas provavelmente você tem 20 ou 30 sistemas embutidos, tais como sistemas de telefones, fogões, micro-ondas etc.

A capacidade de resposta em tempo real é a diferença crítica entre sistemas embutidos e outros sistemas de software, como os sistemas de informações, os sistemas baseados em Web ou os sistemas de software pessoais, cuja principal finalidade é o processamento de dados. Para os sistemas de tempo não real, sua correção pode ser definida especificando-se como as entradas de sistema mapeiam as saídas correspondentes que devem ser produzidas pelo sistema. Em resposta a uma entrada, deve ser gerada uma saída correspondente. Muitas vezes, alguns dados devem ser armazenados. Por exem-

plo, se você escolher um comando 'criar' em um sistema de informações de paciente, a resposta correta de sistema será criar um novo registro de paciente em um banco de dados e confirmar que isso tenha sido feito, dentro de um período razoável, não importa quanto tempo esse processo leve.

Em um sistema de tempo real, a correção depende tanto da resposta para uma entrada quanto do tempo necessário para gerar essa resposta. Se o sistema demorar muito para responder, a resposta necessária poderá ser ineficaz. Por exemplo, se um software embutido muito lento controlar um carro com um sistema de frenagem, um acidente poderá ocorrer porque é impossível parar o carro na hora.

Portanto, o tempo é inerente à definição de um sistema de software de tempo real:

> *Um sistema de software de tempo real é um sistema cujo funcionamento correto depende tanto dos resultados produzidos pelo sistema quanto do tempo em que esses resultados são produzidos. Um 'sistema de tempo real' é um sistema cuja operação é degradada se os resultados não forem produzidos em conformidade com os requisitos de tempo especificados. Se os resultados não forem produzidos de acordo com a especificação de tempo em um 'sistema de tempo real pesado', isso é considerado uma falha de sistema.*

A resposta no tempo certo é um fator importante em todos os sistemas embutidos, mas nem todos os sistemas embutidos exigem uma resposta muito rápida. Por exemplo, o software de bomba de insulina que eu tenho usado como exemplo em vários capítulos deste livro é um sistema embutido. No entanto, embora ele precise verificar o nível de glicose em intervalos periódicos, não precisa responder rapidamente aos eventos externos. O software de estação meteorológica no deserto também é um sistema embutido, mas que não requer uma resposta rápida a eventos externos.

Existem outras diferenças importantes entre sistemas embutidos e outros tipos de sistema de software, além da necessidade de respostas em tempo real:

1. Geralmente, os sistemas embutidos executam continuamente e não param. Eles começam quando o hardware é ligado e devem executar até que o hardware seja desligado. Isso significa que técnicas de engenharia de software confiáveis, como discutido no Capítulo 13, podem precisar ser usadas para garantir a operação contínua. O sistema de tempo real pode incluir mecanismos de atualização que suportam reconfiguração dinâmica para que o sistema possa ser atualizado enquanto está em serviço.
2. As interações com o ambiente do sistema são incontroláveis e imprevisíveis. Em sistemas interativos, o ritmo da interação é controlado pelo sistema e, ao limitar as opções de usuário, os eventos a serem processados são conhecidos antecipadamente. Por outro lado, os sistemas embutidos de tempo real devem ser capazes de responder a eventos inesperados a qualquer momento. Isso gera um projeto de sistemas de tempo real baseado em concorrência, com vários processos executando em paralelo.
3. Podem haver limitações físicas que afetem o projeto de um sistema. Exemplos desse tipo incluem limitações sobre a energia disponível para o sistema e o espaço físico ocupado pelo hardware. Essas limitações podem gerar requisitos para o software embutido, como a necessidade de conservar a energia e, assim, prolongar a vida útil da bateria. Limitações de tamanho e peso podem significar que o software tem de assumir algumas funções de hardware por causa da necessidade de limitar o número de *chips* usados no sistema.
4. A interação direta com o hardware pode ser necessária. Em sistemas interativos e sistemas de informações, existe uma camada de software (os *drivers* de dispositivo) que esconde o hardware do sistema operacional. Isso é possível porque você só pode se conectar a alguns poucos tipos de dispositivos para esses sistemas, como teclados, mouses, monitores etc. Por outro lado, os sistemas embutidos podem ter de interagir com uma ampla gama de dispositivos de hardware que não possuem *drivers* separados de dispositivo.
5. Questões de segurança e confiabilidade podem dominar o projeto de sistema. Muitos sistemas embutidos controlam dispositivos cuja falha pode ter custos humanos ou econômicos elevados. Nesse caso, a confiança é crítica, e o projeto de sistema precisa garantir um comportamento crítico de segurança em todos os momentos. Isso costuma incentivar uma abordagem conservadora para o projeto, em que são usadas técnicas experimentadas e testadas em vez das mais recentes, que podem introduzir novos modos de falhas.

Os sistemas embutidos podem ser considerados reativos; ou seja, eles devem reagir a eventos em seu ambiente com a velocidade desse ambiente (BERRY, 1989; LEE, 2002). Frequentemente, os tempos de resposta são regulados pelas leis da física, e não escolhidos por conveniência humana. Isso está em contraste com outros tipos de software nos quais o sistema controla a velocidade da interação. Por exemplo, o processador de texto que uso para escrever este livro pode verificar a ortografia e a gramática, e não existem limites práticos para o tempo necessário para fazer isso.

20.1 Projeto de sistemas embutidos

O processo de projeto para sistemas embutidos é um processo de engenharia de sistemas em que os projetistas de software devem considerar em detalhes o projeto e o desempenho do hardware de sistema. Parte do processo de projeto do sistema pode envolver e decidir quais recursos de sistema serão implementados no software e no hardware. Os custos e o consumo de energia do hardware são críticos para muitos sistemas de tempo real, embutidos em produtos de consumo, como os sistemas de telefones celulares. Os processadores específicos projetados para oferecer suporte a sistemas embutidos podem ser usados e, para alguns sistemas, um hardware especial pode ter de ser projetado e construído.

Para a maioria dos sistemas de tempo real, um processo de projeto de software de cima para baixo, o qual começa com um modelo abstrato decomposto e desenvolvido em uma série de estágios, é impraticável. Decisões de baixo nível em hardware, software de suporte e o *timing* de sistema devem ser consideradas no início do processo. Tais fatores limitam a flexibilidade dos projetistas de sistema e podem significar que funcionalidade de software adicional, como gerenciamento de bateria e energia, deve ser incluída no sistema.

Os sistemas embutidos são sistemas reativos que reagem a eventos em seu ambiente, e a abordagem geral de projeto de software embutido de tempo real é baseada em um modelo de estímulo-resposta. Um estímulo é um evento que ocorre no ambiente do sistema de software que faz com que o sistema reaja de alguma forma. Uma resposta é um sinal ou mensagem enviada pelo software para seu ambiente.

Você pode definir o comportamento de um sistema real listando os estímulos recebidos pelo sistema, as respostas associadas e o tempo em que a resposta deve ser produzida. A Tabela 20.1 mostra possíveis estímulos e respostas do sistema para um sistema de alarme contra roubo. Na Seção 20.2.1 há mais informações sobre esse sistema.

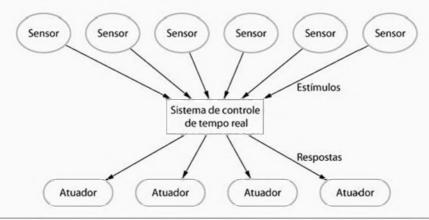
Os estímulos são divididos em duas classes:

1. *Periódicos*. Estímulos que ocorrem em intervalos previsíveis. Por exemplo, o sistema pode examinar um sensor a cada 50 milissegundos (ms) e agir (responder) em função desse valor de sensor (estímulo).
2. *Aperiódicos*. Estímulos que ocorrem de forma irregular e imprevisível e geralmente são sinalizados pelo mecanismo de interrupção do computador. Um exemplo desse estímulo seria uma interrupção indicando que uma transferência de E/S foi concluída e que os dados estavam disponíveis em um *buffer*.

Os estímulos provêm de sensores no ambiente do sistema, e as respostas são enviadas aos atuadores, como mostra a Figura 20.1. Uma diretriz de projeto geral para sistemas de tempo real é ter processos separados para cada tipo de sensor e atuador (Figura 20.2). Esses atuadores controlam equipamentos, como uma bomba, e, em seguida, fazem alterações no ambiente do sistema. Os atuadores também podem gerar estímulos. Os estímulos dos atuadores geralmente indicam que ocorreu algum problema, que deve ser tratado pelo sistema.

Tabela 20.1 Estímulos e respostas para um sistema de alarme contra roubo

Estímulo	**Resposta**
Único sensor positivo	Iniciar o alarme; acender luzes em torno do local do sensor positivo.
Dois ou mais sensores positivos	Iniciar o alarme; acender luzes em torno dos locais de sensores positivos; chamar a polícia com a localização suspeita do arrombamento.
Queda de tensão entre 10 e 20%	Comutar para bateria reserva; executar o teste de fornecimento de energia.
Queda de tensão de mais de 20%	Comutar para bateria reserva; iniciar o alarme; chamar a polícia; executar o teste de fornecimento de energia.
Falha de fonte de alimentação	Chamar o serviço técnico.
Falha de sensor	Chamar o serviço técnico.
Botão de pânico de console positivo	Iniciar alarme; acender luzes em torno do console; chamar a polícia.
Limpar alarmes	Desligar todos os alarmes ativos; apagar todas as luzes que tiverem sido ligadas.

Figura 20.1 O modelo geral de sistemas embutidos de tempo real

Para cada tipo de sensor pode haver um processo de gerenciamento que trata a coleta de dados dos sensores. Os processos de processamento de dados calculam as respostas necessárias para os estímulos recebidos pelo sistema. Os processos de controle de atuadores estão associados com cada atuador e gerenciam sua operação. Esse modelo permite que os dados sejam rapidamente coletados do sensor (antes de ser sobrescrita pela próxima entrada), e que o processamento e a resposta do atuador associado sejam realizados mais tarde.

Um sistema de tempo real precisa responder aos estímulos que ocorrem em momentos diferentes. Portanto, você deve organizar a arquitetura do sistema para que, assim que um estímulo seja recebido, o controle seja transferido para o tratador correto. Em programas sequenciais isso é impraticável. Consequentemente, os sistemas de software de tempo real são normalmente desenvolvidos como um conjunto de processos concorrentes colaborativos. Para apoiar o gerenciamento desses processos, a plataforma de execução que executa o sistema de tempo real pode incluir um sistema operacional de tempo real (discutido na Seção 20.4). As funções fornecidas por esse sistema operacional são acessadas através do sistema de suporte de *run-time* para a linguagem de programação de tempo real usada.

Não há um processo-padrão de projeto de sistemas embutidos. Em vez disso, processos diferentes são usados dependendo do tipo do sistema, do hardware disponível e da organização que está desenvolvendo o sistema. As seguintes atividades podem ser incluídas em um processo de projeto de software de tempo real:

1. *Seleção de plataforma*. Nessa atividade, você escolhe uma plataforma de execução para o sistema (ou seja, o hardware e o sistema operacional de tempo real a ser usado). Os fatores que influenciam essas opções incluem as restrições de tempo sobre o sistema, as limitações de potência disponível, a experiência da equipe de desenvolvimento e o preço-alvo para o sistema entregue.
2. *Identificação de estímulos/resposta*. Isso envolve a identificação dos estímulos que o sistema deve processar e a resposta ou respostas associadas a cada estímulo.
3. *Análise de* timing. Para cada estímulo e resposta associada, identificam-se as restrições de tempo que se aplicam ao estímulo e ao processamento de resposta. Estes são usados para estabelecer *deadlines* para os processos do sistema.

Figura 20.2 Processos de sensores e atuadores

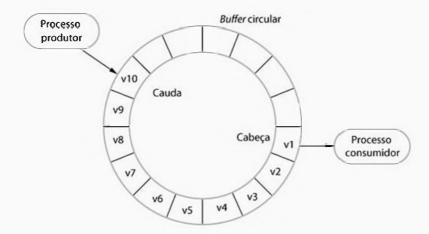
4. *Projeto de processo.* Nesse estágio, agrega-se o estímulo e a transformação da resposta em um número de processos concorrentes. Um bom ponto de partida para projetar a arquitetura de processo são os padrões da arquitetura descritos na Seção 20.2. Em seguida, otimiza-se a arquitetura de processo para refletir os requisitos específicos que se devem implementar.
5. *Projeto de algoritmo.* Para cada estímulo e resposta, criam-se algoritmos para efetuar os processamentos necessários. Projetos de algoritmo podem precisar ser desenvolvidos relativamente cedo no processo de projeto para dar uma indicação da quantidade de processamento e tempo necessários para concluir o processamento. Isso é especialmente importante para tarefas computacionalmente intensivas, como processamento de sinais.
6. *Projeto de dados.* Você especifica as informações que são trocadas por processos e os eventos que coordenam a troca de informações, e cria estruturas de dados para gerenciar essas trocas de informações. Vários processos concorrentes podem compartilhar essas estruturas de dados.
7. *Programação de processo.* Você projeta um sistema de programação que garantirá que os processos são iniciados no tempo certo para cumprirem seus *deadlines*.

A ordem dessas atividades de processo de projeto de software de tempo real depende do tipo de sistema a ser desenvolvido, bem como seus requisitos de processo e plataforma. Em alguns casos, você pode ser capaz de seguir uma abordagem bastante abstrata em que começa com os estímulos e processamentos associados e decide sobre as plataformas de hardware e de execução no final. Em outros casos, a escolha do hardware e do sistema operacional é feita antes de se iniciar o projeto de software. Em tal situação, você precisa projetar o software para levar em conta as restrições impostas pelos recursos de hardware.

Os processos em um sistema de tempo real precisam ser coordenados e compartilhar informações. Os mecanismos de coordenação de processo garantem a exclusão mútua de recursos compartilhados. Quando um processo está modificando um recurso compartilhado, outros processos não devem ser capazes de alterar esse recurso. Os mecanismos para garantir a exclusão mútua incluem semáforos (DIJKSTRA, 1968), monitores (HOARE, 1974) e regiões críticas (BRINCH-HANSEN, 1973). Esses mecanismos de sincronização de processo são descritos na maioria dos textos de sistemas operacionais (SILBERSCHATZ et al., 2008; TANENBAUM, 2007).

Ao se projetar a troca de informações entre os processos, é necessário levar em conta o fato de que esses processos podem estar executando em diferentes velocidades. Um processo está produzindo informações; o outro processo está consumindo essas informações. Se o produtor está sendo executado mais rapidamente que o consumidor, novas informações podem sobrescrever uma informação anterior antes que o processo consumidor leia as informações originais. Se o processo consumidor é executado mais rapidamente do que o processo produtor, o mesmo item pode ser lido duas vezes.

Para contornar esse problema, você deve implementar a troca de informações usando um *buffer* compartilhado e usar mecanismos de exclusão mútua para controlar o acesso a esse *buffer*. Isso significa que uma informação não poderá ser sobrescrita antes de ser lida e que essa informação não poderá ser lida duas vezes. A Figura 20.3 ilustra a noção de um *buffer* compartilhado. Geralmente, ele é implementado como uma fila circular, para que as incompatibilidades de velocidade entre os processos produtor e consumidor possam ser acomodadas sem precisar atrasar a execução de processo.

Figura 20.3 Processos produtor/consumidor compartilhando um *buffer* circular

O processo produtor sempre insere dados na localização do *buffer* na cauda da fila (representada como v10 na Figura 20.3). O processo consumidor sempre recupera informações da cabeça da fila (representada como v1 na Figura 20.3). Após o processo consumidor ter recuperado a informação, a cabeça da lista é ajustada para apontar o próximo item (v2). Depois que o processo produtor adiciona informações, a cauda da lista é ajustada para apontar para o próximo *slot* livre da lista.

Certamente, é importante assegurar que os processos produtor e consumidor não tentem acessar o mesmo item ao mesmo tempo (isto é, quando cabeça = cauda). Você também precisa garantir que o processo produtor não adicione itens a um *buffer* cheio, e que o processo consumidor não retire itens de um *buffer* vazio. Para fazer isso, você pode implementar o *buffer* circular como um processo com operações Get e Put para acessar o *buffer*. A operação Put é chamada pelo processo produtor, e a operação Get, pelo consumidor. Primitivas de sincronização, como semáforos ou regiões críticas, são usadas para garantir que a operação de Get e a de Put sejam sincronizadas, de modo que não acessem o mesmo local ao mesmo tempo. Se o *buffer* estiver cheio, o processo Put precisará esperar até que um *slot* esteja livre; se o *buffer* estiver vazio, o processo Get precisará esperar até que seja feita uma entrada.

Após escolher a plataforma de execução para o sistema, projetar uma arquitetura de processo e decidir-se sobre uma política de programação, talvez você precise verificar se o sistema atenderá a seus requisitos de *timing*. Você pode fazer isso por meio da análise estática do sistema, usando o conhecimento do comportamento de *timing* de componentes, ou por simulação. Essa análise pode revelar que o sistema não funcionará adequadamente. A arquitetura de processo, a política de programação, a plataforma de execução ou todos esses fatores, em seguida, podem precisar ser reprojetados para melhorar o desempenho do sistema.

Algumas vezes, as restrições de *timing* ou outros requisitos podem significar que é melhor implementar algumas funções de sistema no hardware, tal como processamento de sinais. Os componentes modernos de hardware, como FPGAs, são flexíveis e podem ser adaptados a diferentes funções. Os componentes de hardware oferecem um desempenho muito melhor do que o software equivalente. Os gargalos no processamento do sistema podem ser identificados e substituídos pelo hardware, evitando, assim, a otimização cara de softwares.

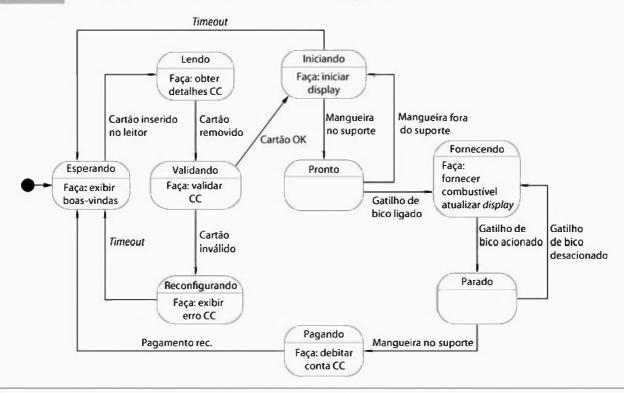
20.1.1 Modelagem de sistema de tempo real

Os eventos aos quais um sistema de tempo real deve reagir costumam fazer com que o sistema se mova de um estado para outro. Por essa razão, modelos de estado, que apresentei no Capitulo 5, são muitas vezes usados para descrever sistemas de tempo real. Um modelo de estado de um sistema pressupõe que, a qualquer momento, o sistema está em um dos vários estados possíveis. Quando um estímulo é recebido, isso pode causar uma transição para um estado diferente. Por exemplo, um sistema controlando uma válvula pode mover-se de um estado 'Válvula aberta' para um estado 'Válvula fechada' quando um comando de operador (estímulo) é recebido.

Os modelos de estado são uma maneira de representar o projeto de um sistema em tempo real, independentemente da linguagem e, portanto, são parte integrante dos métodos de projeto de sistema de tempo real (GOMAA, 1993). A UML suporta o desenvolvimento de modelos de estado baseados em *Statecharts* (HAREL, 1987; HAREL, 1988). *Statecharts* são modelos formais de máquina de estado que suportam estados hierárquicos, para que grupos de estados possam ser considerados uma entidade única. Douglass discute o uso de UML no desenvolvimento de sistemas de tempo real (DOUGLASS, 1999). Os modelos de estado são usados na engenharia dirigida a modelos, discutida no Capítulo 5, para definir a operação de um sistema. Eles podem ser transformados automaticamente em um programa executável.

Essa abordagem para modelagem de sistema foi ilustrada no Capítulo 5, em que usei um exemplo de modelo de um forno de micro-ondas simples. A Figura 20.4 é outro exemplo de um modelo de máquina de estado que mostra o funcionamento de um sistema de software embutido de fornecimento de combustível em uma bomba de gasolina. Os retângulos arredondados representam estados de sistema e as setas representam estímulos que forçam a transição de um estado para outro. Os nomes escolhidos no diagrama de máquina de estado são descritivos. As informações associadas indicam ações tomadas pelos atuadores do sistema ou informações exibidas. Observe que esse sistema nunca se encerra, mas entra em estado de espera quando a bomba não está operando.

O sistema de fornecimento de combustível é projetado para permitir a operação autônoma. O comprador insere um cartão de crédito em um leitor de cartão montado na bomba. Isso faz com que haja transição para um estado de leitura em que os detalhes do cartão são lidos e, em seguida, solicita-se ao comprador que remova o

Figura 20.4 Modelo de máquina de estado de uma bomba de gasolina

cartão. A remoção do cartão aciona uma transição para um estado de validação, em que o cartão é validado. Se o cartão for válido, o sistema inicia a bomba e, quando a mangueira de combustível é removida de seu suporte, transita para o estado de fornecimento, em que ela está pronta para fornecer combustível. Acionar o gatilho no bico faz com que o combustível seja bombeado. O bombeamento para quando o gatilho é desacionado (para simplificar, eu ignorei o interruptor de pressão projetado para parar o derramamento de combustível). Após o fornecimento de combustível estar completo e o comprador ter recolocado a mangueira em seu suporte, o sistema se move para um estado de pagamento, em que a conta do usuário é debitada. Após o pagamento, o software de bomba retorna para o estado de espera.

20.1.2 Programação em tempo real

As linguagens de programação para desenvolvimento de sistemas de tempo real precisam incluir recursos para acessar o hardware de sistema, e deve ser possível prever o *timing* de determinadas operações nessas linguagens. Os sistemas de tempo real pesados ainda, às vezes, são programados em linguagem *assembly* para poder atender aos *deadlines* apertados. São também amplamente usadas linguagens de nível de sistema, como C, que permitem gerar códigos eficientes.

A vantagem de se usar uma linguagem de programação de sistemas como C é que ela permite o desenvolvimento de programas muito eficientes. No entanto, essas linguagens não incluem construções para oferecer suporte à concorrência ou ao gerenciamento de recursos compartilhados. A concorrência e o gerenciamento de recursos são implementados por meio de chamadas para primitivas fornecidas pelo sistema operacional de tempo real, como semáforos para exclusão mútua. Essas chamadas não podem ser verificadas pelo compilador, então é provável que ocorram erros de programação. Frequentemente, os programas têm dificuldade de se entender também porque a linguagem não inclui recursos de tempo real. Além de conhecer o programa, o leitor também precisa saber como o suporte de tempo real é fornecido por meio de chamadas de sistema.

Porque os sistemas em tempo real devem satisfazer suas restrições de *timing*, eles podem não ser capazes de usar o desenvolvimento orientado a objetos para sistemas de tempo real pesados. O desenvolvimento orientado a objetos envolve esconder representações de dados e acessar os valores de atributos por meio de operações definidas com o objeto. Isso significa que existe um significativo *overhead* de desempenho em sistemas orientados

a objetos, porque são necessários códigos extras para mediar o acesso a atributos e tratar as chamadas para operações. A consequente perda de desempenho pode tornar impossível cumprir *deadlines* em tempo real.

Uma versão do Java foi projetada para o desenvolvimento de sistemas embutidos (DIBBLE, 2008), com implementações de diferentes empresas, como IBM e Sun. Essa linguagem inclui um mecanismo para *thread* que permite que *threads* sejam especificados e que não sejam interrompidos pelo mecanismo de coleta de lixo da linguagem. Tratamentos de eventos assíncronos e especificação de *timing* também foram incluídos. No entanto, até o momento, eles foram usados principalmente em plataformas com processador e capacidade de memória significativos (por exemplo, telefones celulares), em vez de sistemas embutidos mais simples com recursos mais limitados. Geralmente, esses sistemas são implementados em C.

20.2 Padrões de arquitetura

Os padrões de arquitetura, apresentados no Capítulo 6, são descrições abstratas e estilizadas de boas práticas de projeto. Eles encapsulam o conhecimento sobre a organização das arquiteturas de sistemas, quando essas arquiteturas devem ser usadas, suas vantagens e desvantagens. Você não deve, no entanto, pensar em um padrão de arquitetura como um projeto genérico para ser instanciado. Em vez disso, use o padrão para entender uma arquitetura e como ponto de partida para criar seu próprio projeto de arquitetura específico.

Como você poderia esperar, as diferenças entre os softwares embutidos e interativos significam que diferentes padrões de arquitetura são usados para sistemas embutidos, em vez de padrões de arquitetura discutidos no Capítulo 6. Os padrões de sistemas embutidos são orientados a processos, em vez de orientadados a objetos e componentes. Nesta seção, discuto três padrões de arquitetura de tempo real comumente usados:

1. *Observar e reagir*. Esse padrão é usado quando um conjunto de sensores é monitorado e exibido rotineiramente. Quando os sensores mostram que ocorreu algum evento (por exemplo, uma chamada recebida em um telefone celular), o sistema reage, iniciando um processo para tratar esse evento.
2. *Controle de ambiente*. É usado quando um sistema inclui sensores que fornecem informações sobre o ambiente e atuadores capazes de alterar o ambiente. Em resposta às mudanças ambientais detectadas pelo sensor, sinais de controle são enviados para os atuadores de sistema.
3. Pipeline *de processo*. Esse padrão é usado quando dados precisam ser transformados de uma representação para outra antes que possam ser processados. A transformação é implementada como uma sequência de etapas de processamento, que podem ser realizadas concorrentemente. Isso permite o processamento de dados muito rápido, porque um núcleo separado ou processador pode executar cada transformação.

Esses padrões podem ser combinados naturalmente e, muitas vezes, você verá mais de um em um único sistema. Por exemplo, quando é usado o padrão 'Controle de ambiente', é muito comum que os atuadores sejam monitorados usando o padrão 'Observar e Reagir'. No caso de uma falha de atuador, o sistema pode reagir exibindo uma mensagem de aviso, desligando o atuador, comutando para um sistema de *backup* etc.

Os padrões que abordo aqui são padrões de arquitetura que descrevem a estrutura geral de um sistema embutido. Douglass (2002) descreve padrões de projeto de tempo real, de baixo nível, usados para ajudá-lo a tomar decisões de projeto mais detalhadas. Esses padrões incluem padrões de projeto para o controle de execução, comunicação, alocação de recursos e segurança e confiabilidade.

Esses padrões de arquitetura devem ser o ponto de partida para um projeto de sistemas embutidos. No entanto, eles não são *templates* de projeto. Se os usar como tal, você provavelmente acabará com uma arquitetura de processo ineficiente. Portanto, é necessário otimizar a estrutura de processo para garantir que você não tem muitos processos. Você também deve garantir que existe uma correspondência clara entre os processos e os sensores e atuadores do sistema.

20.2.1 Observar e Reagir

Os sistemas de monitoração são uma importante classe dos sistemas embutidos de tempo real. Um sistema de monitoração examina seu ambiente por meio de um conjunto de sensores e, geralmente, exibe o estado do ambiente de alguma forma. Isso poderia ser em uma tela interna, em *displays* especiais ou em um *display* remoto. Se algum evento ou estado do sensor excepcional é detectado pelo sistema, o sistema de monitoração inicia algu-

ma ação. Muitas vezes, isso envolve ligar um alarme para chamar a atenção do operador para o evento. Às vezes, o sistema pode dar início a outras ações preventivas, tais como desligar o sistema para preservá-lo de danos.

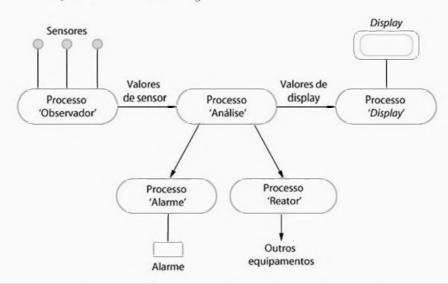
O padrão 'Observar e Reagir' (Tabela 20.2 e Figura 20.5) é usado em sistemas de monitoração. Os valores dos sensores são observados, e quando são detectados valores específicos, o sistema reage de alguma forma. Os sistemas de monitoração podem ser compostos de várias instâncias do padrão 'Observar e reagir', uma para cada tipo de sensor no sistema. Dependendo dos requisitos de sistema, você pode otimizar o projeto pela combinação de processos (por exemplo, você pode usar um único processo de *display* para exibir as informações de todos os diferentes tipos de sensores).

Como exemplo de uso desse padrão, considere o projeto de um sistema de alarme contra roubo que pode ser instalado em um prédio de escritórios:

> *Um sistema de software é implementado como parte de um sistema de alarme contra roubo em edifícios comerciais. Ele usa vários tipos de sensores, incluindo detectores de movimento em salas individuais, sensores de porta que detectam a abertura de portas do corredor e, no térreo, sensores de janela, que podem detectar quando uma janela foi aberta.*
>
> *Quando um sensor detecta a presença de um intruso, o sistema automaticamente chama a polícia local e, usando um sintetizador de voz, relata a localização do alarme. Ele alterna as luzes nas salas em torno do sensor ativo e dispara um alarme sonoro. Normalmente, o sistema de sensor é alimentado por uma corrente elétrica, mas é equipado com uma bateria reserva. A perda de energia é detectada com o uso de um monitor de circuito de alimentação separado, que monitora a tensão da rede. Se é detectada uma queda na tensão, o sistema pressupõe que intrusos interromperam o fornecimento de energia e, assim, um alarme é disparado.*

Tabela 20.2 O padrão 'Observar e Reagir'

Nome	Observar e Reagir
Descrição	Os valores de entrada de um conjunto de sensores de mesmo tipo são coletados e analisados. Esses valores são exibidos de alguma forma. Se os valores de sensor indicam que alguma condição excepcional surgiu, são tomadas ações para chamar a atenção do operador para esse valor e, em certos casos, ações em resposta ao valor excepcional.
Estímulos	Valores de sensores conectados ao sistema.
Respostas	Saídas para *display*, acionadores de alarmes, sinais para sistemas de reação.
Processos	Observador, Análise, *Display*, Alarme, Reator.
Usado em	Sistemas de monitoração, sistemas de alarme.

Figura 20.5 Estrutura de processo 'Observar e Reagir'

Uma possível arquitetura de processo para o sistema de alarme é mostrada na Figura 20.6. Nesse diagrama, as setas representam sinais enviados de um processo para outro. Esse sistema é um sistema de tempo real 'leve' que não tem requisitos de *timing* rigorosos. Os sensores não precisam detectar eventos de alta velocidade, de modo que eles só precisam ser monitorados, relativamente, com pouca frequência. Os requisitos de tempo para esse sistema são abordados na Seção 20.3.

Na Tabela 20.1, apresentei os estímulos e respostas desse sistema de alarme. Eles são usados como ponto de partida para o projeto de sistema. O padrão 'Observar e Reagir' é usado nesse projeto. Existem processos 'Observador' associados com cada tipo de sensor, e processos 'Reator' para cada tipo de reação. Existe um processo de análise único que verifica os dados de todos os sensores. Os processos de exibição no padrão são combinados em um único processo de *display*.

20.2.2 Controle de ambiente

Talvez o maior uso dos softwares embutidos seja em sistemas de controle. Nesses sistemas, o software controla a operação dos equipamentos com base em estímulos do ambiente do equipamento. Por exemplo, um sistema de frenagem antiderrapante em um carro monitora as rodas do carro e o sistema de freio (ambiente do sistema). Ele procura sinais de que as rodas estão derrapando quando a pressão do freio é aplicada. Se esse for o caso, o sistema ajusta a pressão do freio a parar o travamento das rodas e reduzir a probabilidade de uma derrapagem.

Os sistemas de controle podem fazer uso do padrão 'Controle de ambiente', que é um padrão de controle geral que inclui processos de sensores e atuadores. Esse padrão é descrito na Tabela 20.3, com a arquitetura de processo mostrada na Figura 20.7. Uma variante desse padrão exclui o processo de *display*. Essa variante é usada em situ-

Figura 20.6 Estrutura de processo de um sistema de alarme contra roubo

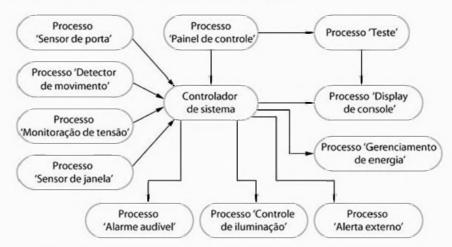
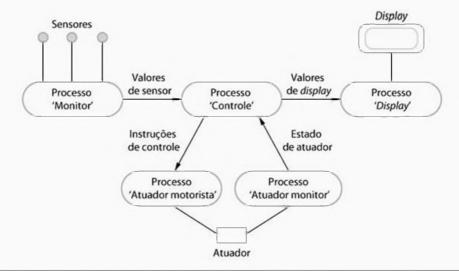



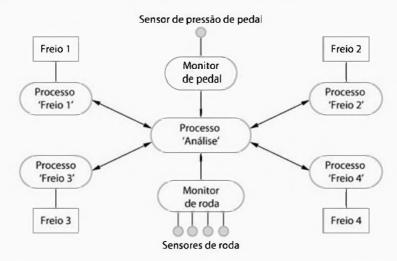
Tabela 20.3 O padrão 'Controle de ambiente'

Nome	**Controle de ambiente**
Descrição	O sistema analisa informações de um conjunto de sensores que coletam dados do ambiente do sistema. Mais informações também podem ser coletadas sobre o estado dos atuadores que também estão conectados. Com base nos dados dos sensores e atuadores, sinais de controle são enviados aos atuadores que, então, causam alterações no ambiente do sistema. Podem ser exibidas informações sobre os valores de sensor e o estado dos atuadores.
Estímulos	Valores de sensores ligados ao sistema e o estado dos atuadores de sistema.
Respostas	Sinais de controle para atuadores, informações de *display*.
Processos	Monitor, Controle, *Display*, Atuador motorista, Atuador monitor.
Usado em	Sistemas de controle.

Figura 20.7 Estrutura de processo 'Controle de ambiente'

ações nas quais não há requisito de intervenção do usuário ou sempre que a taxa de controle é tão alta que um *display* não seria significativo.

Esse padrão pode ser a base do projeto de um sistema de controle com uma instanciação do padrão 'Controle de ambiente' para cada atuador (ou tipo de atuador) que está sendo controlado. Em seguida, você otimiza o projeto para reduzir o número de processos. Por exemplo, você pode combinar a monitoração de atuador e os processos de controle de atuador, ou pode ter um único processo de monitoração e controle para vários atuadores. As otimizações que você escolhe dependem dos requisitos de *timing*. Talvez você precise monitorar sensores mais frequentemente do que enviar sinais de controle, casos em que pode ser impraticável combinar o controle e a monitoração de processos. Também pode haver *feedback* direto entre o controle de atuador e o processo de monitoração de atuador. Isso permite que sejam tomadas decisões de controle de baixa granularidade pelo processo de controle de atuador.

Na Figura 20.8 você pode ver como esse padrão funciona. Ela mostra o exemplo de um controlador para um sistema de frenagem de um carro. O ponto de partida para o projeto é associar uma instância do padrão de cada tipo de atuador do sistema. Nesse caso, existem quatro atuadores, cada um controlando o freio de uma roda. Os processos de sensores individuais são combinados em um único processo de monitoração de rodas que analisa os sensores em todas as rodas. Ele monitora o estado de cada roda para verificar se a roda está travada ou se está girando. Um processo separado monitora a pressão sobre o pedal de freio exercida pelo motorista do carro.

Figura 20.8 Arquitetura de sistema de controle de um sistema de frenagem antiderrapante

O sistema inclui um recurso antiderrapante, ativado se os sensores indicam que uma roda é travada quando o freio é acionado. Isso significa que não existe atrito suficiente entre a via e o pneu. Em outras palavras, o carro está derrapando. Se a roda está bloqueada, o motorista não pode controlá-la. Para compensar isso, o sistema envia uma sequência rápida de sinais de ligar/desligar para o freio da roda, que permite à roda girar e recuperar o controle.

20.2.3 Processo *Pipeline*

Muitos sistemas de tempo real estão preocupados com a coleta de dados de ambiente do sistema, os quais transformam os dados de sua representação original em outra representação digital que pode ser mais facilmente analisada e processada pelo sistema. O sistema também pode converter dados digitais de dados analógicos, e depois envia para seu ambiente. Por exemplo, um rádio de software aceita a entrada de pacotes de dados digitais que representa a transmissão de rádio e os transforma em um sinal sonoro que as pessoas podem ouvir.

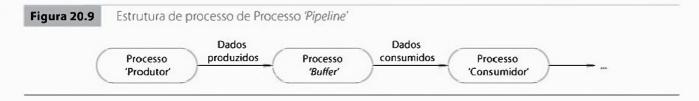
O processamento de dados que estão envolvidos em muitos desses sistemas precisa ser efetuado muito rapidamente. Caso contrário, os dados de entrada podem ser perdidos e os sinais de saída podem ser quebrados por falta de informações essenciais. O padrão Processo *Pipeline* possibilita esse processamento rápido ao quebrar o processamento de dados necessários em uma sequência de transformações separadas, com cada transformação efetuada por um processo independente. Essa é uma arquitetura muito eficiente para sistemas que usam vários processadores ou processadores de múltiplos núcleos. Cada processo no *pipeline* pode ser associado a um processador ou núcleo separado para que as etapas de processamento possam funcionar em paralelo.

A Tabela 20.4 é uma breve descrição do padrão *pipeline* de dados, e a Figura 20.9 mostra a arquitetura de processo para esse padrão. Observe que os processos envolvidos podem produzir e consumir informações. Eles são ligados por *buffers* sincronizados, conforme discutido na Seção 20.1. Isso permite que os processos produtor e consumidor operem em velocidades diferentes, sem perda de dados.

Um exemplo de sistema que pode usar um processo *pipeline* é um sistema de aquisição de dados em alta velocidade. Sistemas de aquisição de dados coletam dados de sensores para análise e processamentos subsequentes. Esses sistemas são usados em situações nas quais os sensores estão coletando uma grande quantidade de dados do ambiente do sistema e não é possível ou necessário processar esses dados em tempo real. Em vez disso, estes são coletados e armazenados para análise posterior. Os sistemas de aquisição de dados são usados em experimentos científicos e sistemas de controle de processo em que os processos físicos, como reações químicas, são muito rápidos. Nesses sistemas, os sensores podem gerar dados muito rapidamente e o sistema de aquisição de dados precisa garantir que uma leitura do sensor seja coletada antes das alterações no valor do sensor.

Tabela 20.4 O padrão Processo *Pipeline*

Nome	**Processo *Pipeline***
Descrição	Um *pipeline* de processos é criado com uma sequência de dados movendo-se de uma das extremidades do *pipeline* para outra. Muitas vezes, os processos são ligados por *buffers* sincronizados para permitir que os processos produtor e consumidor sejam executados em velocidades diferentes. O auge de um *pipeline* pode ser armazenamento ou exibição de dados ou o *pipeline* pode terminar em um atuador.
Estímulos	Valores de entrada de ambiente ou de outro processo.
Respostas	Valores de saída para o ambiente ou um *buffer* compartilhado.
Processos	Produtor, *Buffer*, Consumidor.
Usado em	Sistemas de aquisição de dados, sistemas multimídia.

Figura 20.9 Estrutura de processo de Processo *'Pipeline'*

A Figura 20.10 é o modelo simplificado de um sistema de aquisição de dados que poderia ser parte do software de controle em um reator nuclear. Trata-se de um sistema que coleta os dados dos sensores de monitoração do fluxo de nêutrons (a densidade de nêutrons) no reator. Os dados de sensor são colocados em um *buffer* do qual são extraídos e processados. O nível médio do fluxo é exibido no *display* do operador e armazenado para futuro processamento.

20.3 Análise de *timing*

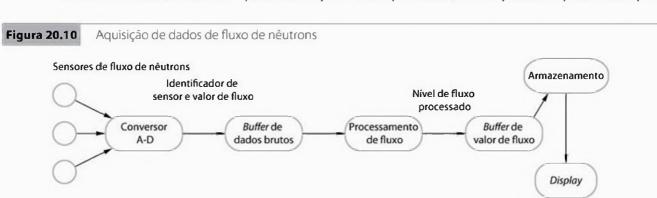
Conforme discutido no início deste capítulo, a correção de um sistema de tempo real depende não apenas da correção de suas saídas, mas também do momento em que essas saídas foram produzidas. Isso significa que uma das atividades importantes no processo de desenvolvimento de software embutido de tempo real é a análise de *timing*. Em uma análise desse tipo, calcula-se com que frequência cada processo do sistema deve ser executado para garantir que todas as entradas sejam processadas e todas as respostas do sistema sejam produzidas no tempo correto. Os resultados da análise de *timing* são usados para decidir quão frequentemente cada processo deve executar e como esses processos devem ser programados pelo sistema operacional de tempo real.

A análise de *timing* para os sistemas de tempo real é particularmente difícil quando os sistemas devem lidar com uma mistura de estímulos periódicos e aperiódicos e repostas. Sendo os estímulos aperiódicos imprevisíveis, é necessário fazer suposições sobre a probabilidade de eles ocorrerem e, portanto, requerem serviços a qualquer momento. Esses pressupostos podem estar incorretos, pois o desempenho do sistema após a entrega pode não ser adequado. O livro de Cooling (2003) discute as técnicas para a análise de desempenho de sistema de tempo real que leva em consideração eventos aperiódicos.

No entanto, como computadores ficaram mais rápidos, tornou-se possível o projeto de muitos sistemas usando apenas estímulos periódicos. Quando os processadores eram lentos, os estímulos aperiódicos precisavam ser usados para garantir que eventos críticos fossem processados antes de seu *deadline*, pois, geralmente, os atrasos no processamento envolviam alguma perda para o sistema. Por exemplo, a falha de uma fonte de alimentação em um sistema embutido pode significar que o sistema precisa desligar o equipamento conectado de forma controlada, dentro de um tempo muito curto (digamos 50 ms). Isso poderia ser implementado como uma interrupção de 'falha de energia elétrica'. No entanto, também pode ser implementado usando um processo periódico que é executado com muita frequência e verifica a energia. Como o tempo entre invocações do processo é curto, ainda existe tempo para realizar um desligamento controlado do sistema antes que a falta de energia provoque danos. Por essa razão, eu centro a discussão nos problemas de *timing* para processos periódicos.

Quando você estiver analisando os requisitos de *timing* dos sistemas embutidos de tempo real e projetar sistemas para atender a esses requisitos, existem três fatores-chave que você deve considerar:

1. *Deadlines*. O tempo no qual os estímulos devem ser processados, e o sistema, produzir alguma resposta. Se o sistema não cumprir um *deadline*, caso seja um sistema de tempo real pesado, essa será uma falha de sistema; em um sistema de tempo real leve, isso indica que o serviço de sistema está degradado.
2. *Frequência*. O número de vezes por segundo que um processo deve executar o serviço para ter certeza de que o sistema pode cumprir seus *deadlines*.
3. *Tempo de execução*. O tempo necessário para processar um estímulo e produzir uma resposta. Muitas vezes, é necessário considerar dois tempos de execução — o tempo médio de execução de um processo e o pior

Figura 20.10 Aquisição de dados de fluxo de nêutrons

tempo de execução para esse processo. O tempo de execução nem sempre é o mesmo por causa da execução condicional do código, dos atrasos em espera por outros processos etc. Em um sistema de tempo real pesado, talvez seja necessário fazer suposições com base no pior tempo de execução para garantir que os *deadlines* não sejam perdidos. Em sistemas de tempo real leves, os cálculos podem ser baseados no tempo médio de execução.

Para continuar o exemplo de uma falha de fornecimento de energia, vamos supor que, após um evento de falha, demora 50 ms para a voltagem cair para um nível em que o equipamento possa ser danificado. Portanto, o processo de desligamento do equipamento deve começar dentro de 50 ms a partir de um evento de falha de energia. Nesses casos, seria prudente definir um *deadline* mais curto, de 40 ms, por causa das variações físicas no equipamento. Isso significa que as instruções de desligamento para todos os equipamentos conectados que correm esse risco devem ser emitidas e processadas dentro de 40 ms, assumindo que o equipamento também depende das falhas na fonte de alimentação.

Se, pela monitoração da voltagem, for detectada a falha de energia, é necessário fazer mais que uma observação para perceber que a tensão está caindo. Se você executar o processo 250 vezes por segundo, significa que ele é executado a cada 4 ms e pode demorar até dois períodos para detectar a queda de tensão. Portanto, é preciso até 8 ms para detectar o problema. Consequentemente, o pior tempo de execução do processo de desligamento não deve exceder 16 ms e garantir que o *deadline* de 40 ms seja atendido. Esse valor é calculado subtraindo-se os períodos de processo (8 ms) do *deadline* (40 ms) e dividindo-se o resultado por dois, pois são necessárias duas execuções de processo.

Na realidade, você normalmente visaria a algum tempo consideravelmente inferior a 16 ms para ter uma margem de segurança, para o caso de seus cálculos estarem errados. Na verdade, o tempo necessário para examinar um sensor e verificar que não houve nenhuma perda significativa de tensão deve ser muito menor do que 16 ms. Isso envolve apenas uma simples comparação entre dois valores. O tempo médio de execução do processo de monitoração de energia deve ser menor que 1 ms.

O ponto de partida para a análise de *timing* em um sistema de tempo real são os requisitos de *timing*, que devem estabelecer os *deadlines* para cada resposta necessária no sistema. A Tabela 20.5 mostra os possíveis requisitos de *timing* para o sistema de alarme contra roubo para prédios de escritório discutido na Seção 20.2.1. Para simplificar esse exemplo, vamos ignorar os estímulos gerados por procedimentos de testes do sistema e sinais externos para redefinir o sistema no caso de um alarme falso. Isso significa que existem apenas dois tipos de estímulos para serem processados pelo sistema:

1. *Falha de energia*. É detectada observando-se uma queda de tensão de mais de 20%. A resposta requerida é comutar o circuito para energia de *backup*, sinalizando um dispositivo eletrônico de comutação de energia, que comuta a fonte principal para o *backup* de bateria.
2. *Alarme de intrusão*. É um estímulo gerado por um sensor do sistema. A resposta a esse estímulo é calcular o número da sala do sensor ativo, configurar uma chamada à polícia, iniciar o sintetizador de voz para gerenciar a chamada e ligar o alarme sonoro que indica a presença de intrusos e as luzes de edifício na área.

Tabela 20.5 Requisitos de *timing* para o sistema de alarme contra roubo

Estímulo/Resposta	Requisitos de timing
Falha de energia	A comutação para o *backup* de energia deve ser concluída no *deadline* de 50 ms.
Alarme de porta	Cada alarme de porta deve ser varrido duas vezes por segundo.
Alarme de janela	Cada alarme de janela deve ser varrido duas vezes por segundo.
Detector de movimento	Cada detector de movimento deve ser varrido duas vezes por segundo.
Alarme sonoro	O alarme sonoro deve ser ligado em meio segundo de um alarme que está sendo gerado por um sensor.
Interruptor de luzes	As luzes devem ser ligadas em meio segundo do alarme que está sendo gerado por um sensor.
Comunicações	A chamada para a polícia deve ser iniciada em dois segundos do alarme que está sendo gerado por um sensor.
Sintetizador de voz	Uma mensagem sintetizada deve estar disponível em dois segundos do alarme que está sendo gerado por um sensor.

Como mostra a Tabela 20.5, você deve listar as restrições de *timing* para cada classe de sensor separadamente, mesmo quando (como no caso presente) são os mesmos. Por considerá-los separadamente, você deixa margem para mudanças e facilita o cálculo do número de vezes que o processo de controle precisa ser executado a cada segundo.

Atribuir as funções do sistema para processos concorrentes é o próximo estágio do projeto. Existem quatro tipos de sensores que devem ser varridos periodicamente, cada um com um processo associado. São eles: o sensor de tensão, sensores de porta, sensores de janelas e detectores de movimento. Normalmente, os processos relacionados com o sensor executarão muito rapidamente, pois o que todos eles estão fazendo é verificar se um sensor mudou seu *status* (por exemplo, de desligado para ligado). É razoável supor que o tempo de execução para verificar e avaliar o estado de um sensor não seja mais que 1 ms.

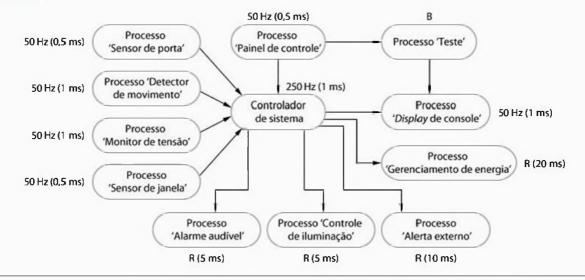
Para garantir o cumprimento dos *deadlines* definidos pelos requisitos de *timing*, você frequentemente precisa decidir como os processos relacionados devem executar e quantos sensores devem ser examinados durante cada execução do processo. Existem compromissos óbvios entre a frequência e o tempo de execução:

1. Se você examinar um sensor durante cada execução de processo e se houver N sensores de determinado tipo, você deve programar os processos para 4N vezes por segundo para garantir que o *deadline* de detecção de uma alteração de estado em 0,25 segundos será cumprido.
2. Se você examinar quatro sensores durante cada execução de processo, o tempo de execução é aumentado para 4 ms, mas você só precisa executar o processo N vezes/segundo para atender ao requisito de *timing*.

Nesse caso, como os requisitos de sistema definem ações quando dois ou mais sensores são positivos, é sensato examinar sensores em grupos, com grupos baseados na proximidade física dos sensores. Se um invasor entrou no edifício, provavelmente os sensores adjacentes serão positivos.

Ao concluir a análise de *timing*, você pode anotar no modelo de processo as informações sobre a frequência de execução e o tempo esperado da execução (ver Figura 20.11 como exemplo). Aqui, processos periódicos são anotados com sua frequência; processos que são iniciados em resposta a um estímulo são anotados com R (do inlgês, *response*) e o processo de teste é um processo em segundo plano, anotado com B ('segundo plano', ou do inglês, *background*). Isso significa que ele só é executado quando o tempo do processador está disponível. Em geral, é mais simples projetar um sistema de modo que contenha um pequeno número de frequências de processos. Os tempos de execução representam os piores tempos de execução dos processos requeridos.

A etapa final do processo de projeto é projetar um sistema de programação que garantirá que um processo sempre será programado para cumprir seus *deadlines*. Só faça isso se você souber as abordagens de programação que são suportadas pelo sistema operacional de tempo real (BURNS e WELLINGS, 2009). O programador do sistema operacional de tempo real aloca um processo para um processador para determinado período de tempo. O tempo pode ser fixo ou pode variar de acordo com a prioridade do processo.

Figura 20.11 *Timing* de processo de alarme

Na atribuição de prioridades de processo, você deve considerar os *deadlines* de cada processo para que os processos com *deadlines* curtos recebam tempo de processador para cumprir o prazo. Por exemplo, o processo monitor de tensão no alarme de roubo precisa ser programado para que quedas de tensão sejam detectadas e uma comutação seja feita para a energia de *backup* antes que o sistema falhe. Isso, portanto, deve ter uma prioridade mais alta do que os processos que verificam os valores dos sensores, pois estes têm um *deadline* menos rigoroso em relação a seu tempo de execução esperado.

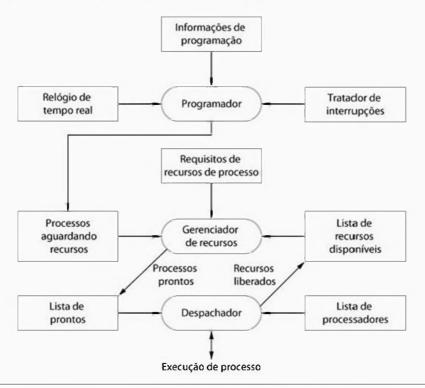
20.4 Sistemas operacionais de tempo real

A plataforma de execução para a maioria dos sistemas de aplicações é um sistema operacional que gerencia recursos compartilhados e fornece recursos como um sistema de arquivos, gerenciamento *run-time* de processos etc. No entanto, a ampla funcionalidade em um sistema operacional convencional ocupa muito espaço e retarda o funcionamento dos programas. Além disso, os recursos de gerenciamento de processo no sistema não podem ser projetados para permitir controle de baixa granularidade sobre o agendamento de processos.

Por essas razões, os sistemas operacionais padrão, como Linux e Windows, normalmente não são usados como a plataforma de execução para sistemas de tempo real. Sistemas embutidos muito simples podem ser implementados como sistemas '*bare metal*' (sem sistema operacional). Os próprios sistemas incluem inicialização e desligamento do sistema, gerenciamento de processo e recursos e programação de processos. Mais comumente, no entanto, os aplicativos embutidos são construídos sobre um sistema operacional de tempo real (RTOS, do inglês *real-time operating system*), que é um sistema operacional eficiente que oferece os recursos necessários para sistemas em tempo real. Alguns exemplos de RTOS são Windows/CE, Vxworks e RTLinux.

Um sistema operacional de tempo real gerencia os processos e a alocação de recursos para um sistema de tempo real. Ele inicia e para os processos para que os estímulos possam ser tratados e aloca os recursos de memória e processador. Os componentes de um RTOS (Figura 20.12) dependem do tamanho e da complexidade do sistema de tempo real que está sendo desenvolvido. Todos os sistemas, exceto os mais simples, geralmente incluem:

1. Um relógio de tempo real, que fornece as informações necessárias para programar os processos periodicamente.
2. Um tratador de interrupções, que gerencia solicitações aperiódicas de serviço.

Figura 20.12 Componentes de um sistema operacional de tempo real

3. Um programador, responsável por examinar os processos que podem ser executados e escolher um desses para execução.
4. Um gerenciador de recursos, que aloca os recursos de memória e processador adequados para processos que foram programados para execução.
5. Um despachador, responsável por iniciar a execução dos processos.

Os sistemas operacionais de tempo real para grandes sistemas, como sistemas de controle de processos ou de telecomunicações, podem ter recursos adicionais, ou seja, gerenciamento de discos de armazenamento, recursos de gerenciamento de defeitos que detectam e relatam defeitos de sistema e um gerente de configuração que ofereça suporte à reconfiguração dinâmica de aplicações de tempo real.

20.4.1 Gerenciamento de processos

Sistemas em tempo real precisam lidar rapidamente com eventos externos e, em alguns casos, cumprir *deadlines* para processamento desses eventos. Isso significa que os processos de tratamento de eventos devem ser programados para execução em tempo de detectar tais eventos. Eles também devem ter alocados recursos de processador suficientes para cumprir seu *deadline*. O gerente de processos em um RTOS é responsável pela escolha dos processos para execução, alocação de recursos de processador e memória, bem como iniciação e parada de execução de processo em um processador.

O gerente de processos precisa gerenciar processos com prioridades diferentes. Para alguns estímulos, como aqueles associados com certos eventos excepcionais, é essencial que seu processamento seja concluído dentro dos limites de tempo especificados. Outros processos podem ser atrasados com segurança se um processo mais crítico requisitar o serviço. Consequentemente, o RTOS precisa ser capaz de gerenciar, pelo menos, dois níveis de prioridade para os processos de sistema:

1. *Nível de interrupção*. É o nível de prioridade mais alto. Ele é alocado para processos que exigem uma resposta muito rápida. Um desses processos será o processo de relógio de tempo real.
2. *Nível de relógio*. Esse nível de prioridade é alocado para os processos periódicos.

Pode haver outro nível de prioridade alocado para processos em segundo plano (tais como um processo de autoverificação) que não precisam cumprir *deadlines* de tempo real. Esses processos são programados para execução quando a capacidade de processador está disponível.

Dentro de cada um desses níveis de prioridade, diferentes classes de processo podem ser alocadas para diferentes prioridades. Por exemplo, pode haver várias linhas de interrupção. Uma interrupção de um dispositivo muito rápido pode ter de parar o processamento de uma interrupção de um dispositivo mais lento para evitar a perda de informações. Normalmente, a alocação de prioridades de processos para que todos sejam atendidos a tempo exige extensa análise e simulação.

Os processos periódicos devem ser executados em intervalos de tempo específicos para o controle de aquisição de dados e atuadores. Em sistemas de tempo real, ocorrerão vários tipos de processos periódicos. Usando os requisitos de *timing* especificados no programa de aplicação, o RTOS organiza a execução dos processos periódicos para que eles possam cumprir seus *deadlines*.

As ações tomadas pelo sistema operacional para o gerenciamento de processos periódicos são mostradas na Figura 20.13. O programador examina a lista de processos periódicos e seleciona um processo para ser executado. A escolha depende da prioridade do processo, dos períodos do processo, dos tempos de execução esperados e dos *deadlines* dos processos prontos. Às vezes, dois processos com *deadlines* diferentes devem ser executados no

Figura 20.13 Ações de RTOS necessárias para iniciar um processo

mesmo tique de relógio. Em tal situação, um processo deve ser atrasado. Geralmente, o sistema escolherá retardar o processo com o *deadline* de mais longo prazo.

Processos que precisam responder rapidamente a eventos assíncronos podem ser dirigidos por interrupção. O mecanismo de interrupção do computador faz com que o controle seja transferido para uma posição de memória predeterminada. Essa posição contém uma instrução de *jump* para uma rotina de serviço de interrupção rápida e simples. A rotina de serviço desabilita outras interrupções para evitar interrupção. Em seguida, ela descobre a causa da interrupção e inicia, com alta prioridade, um processo para tratar o estímulo causador da interrupção. Em alguns sistemas de aquisição de dados em alta velocidade, o tratador de interrupções salva os dados que a interrupção sinalizou como disponíveis em um *buffer* para processamento posterior. Então, as interrupções são habilitadas novamente e o controle retorna para o sistema operacional.

A qualquer momento, podem ser executados vários processos, todos com prioridades diferentes. O programador de processos implementa políticas de programação de sistema que determinam a ordem de execução de processos. Existem duas estratégias de programação usadas com frequência:

1. *Programação não preemptiva.* Uma vez que um processo tenha sido programado para execução, ele é executado até ser concluído ou até que ele seja bloqueado por alguma razão, tal como esperar por uma entrada. No entanto, isso pode causar problemas quando houver processos com prioridades diferentes e um processo de alta prioridade tenha de esperar o fim de um processo de baixa prioridade.
2. *Programação preemptiva.* A execução de um processo em execução pode ser interrompida caso um processo de prioridade mais alta requisite serviço. Os processos de prioridade mais alta param a execução do processo de prioridade mais baixa e são alocados em um processador.

Dentro dessas estratégias, desenvolveram-se diferentes algoritmos de programação. Estes incluem: programação *round-robin*, em que um processo é executado de cada vez; programação *rate monotonic*, na qual há prioridade para o processo com o período mais curto (frequência mais alta); e programação *shortest deadline first*, em que é programado o processo na fila com o *deadline* mais curto (BURNS e WELLINGS, 2009).

Informações sobre o processo a ser executado são passadas para o gerenciador de recursos. O gerenciador de recursos aloca memória e, em um sistema com vários processadores, também adiciona um processador para esse processo. O processo é, então, colocado na 'lista de prontos' — uma lista de processos prontos para execução. Quando um processador termina a execução de um processo e este é disponibilizado, o despachador é invocado. Ele examina a lista de prontos para encontrar um processo que possa ser executado no processador disponível e inicia sua execução.

- Um sistema embutido de software é parte de um sistema de hardware/software que reage a eventos em seu ambiente. O software é 'embutido' no hardware. Sistemas embutidos são normalmente sistemas de tempo real.
- Um sistema de tempo real é um sistema de software que deve responder a eventos em tempo real. A correção do sistema não depende apenas dos resultados que produz, mas também do tempo em que esses resultados são produzidos.
- Geralmente, os sistemas de tempo real são implementados como um conjunto de processos de comunicação que reagem a estímulos para produzir respostas.
- Os modelos de estado são uma importante representação de projeto para sistemas embutidos de tempo real. Eles são usados para mostrar como o sistema reage ao ambiente na medida em que eventos desencadeiam mudanças no estado do sistema.
- Existem vários padrões que podem ser observados em diferentes tipos de sistemas embutidos. Eles incluem um padrão para monitoração de ambiente do sistema para eventos adversos, um padrão para o controle de atuadores e um padrão de processamento de dados.
- Os projetistas de sistemas de tempo real precisam fazer uma análise de *timing*, dirigida pelos *deadlines* de processamento e resposta a estímulos. Eles precisam decidir quantas vezes cada processo no sistema deve ser executado, o tempo de execução esperado e o pior caso de tempo de execução do processo.
- Um sistema operacional em tempo real é responsável pelo gerenciamento de processos e recursos. Ele sempre inclui um programador, que é o componente responsável por decidir qual processo deve ser programado para execução.

LEITURA COMPLEMENTAR

Software Engineering for Real-Time Systems. Escrito a partir de uma perspectiva de engenharia em vez de uma perspectiva de ciência de computação, esse livro é um bom guia prático para a engenharia de sistemas de tempo real. Ele tem uma boa cobertura sobre questões de hardware, então é um excelente complemento ao livro de Burns e Wellings (veja adiante). COOLING, J. *Software Engineering for Real-Time Systems.* Addison-Wesley, 2003.)

Real-time Systems and Programming Language: Ada, Real-time Java and C/Real-time POSIX, 4th edition. Um texto excelente e abrangente que fornece ampla cobertura de todos os aspectos de sistemas de tempo real. (BURNS, A.; WELLINGS, R. *Real-time Systems and Programming Language*: Ada, Real-time Java and C/Real-time POSIX. 4. ed. Addison-Wesley, 2009.)

'Trends in Embedded Software Engineering'. Esse artigo sugere que o desenvolvimento dirigido a modelos (como discutido no Capítulo 5) se tornará uma abordagem importante para o desenvolvimento de sistemas embutidos. Ele é parte de uma edição especial sobre sistemas embutidos, e você pode achar que os outros artigos também são uma leitura útil. (*IEEE Software*, v. 26, n. 3, mai.-jun. 2009.) Disponível em: <http://dx.doi.org/10.1109/MS.2009.80>.

EXERCÍCIOS

20.1 Usando exemplos, explique por que, geralmente, os sistemas de tempo real precisam ser implementados usando processos concorrentes.

20.2 Identifique os possíveis estímulos e as respostas esperadas para um sistema embutido que controla uma geladeira doméstica ou uma máquina de lavar roupa doméstica.

20.3 Usando a abordagem baseada em estados para modelagem, conforme discutido na Seção 20.1.1, modele a operação de um sistema embutido de software para um sistema de caixa postal de voz de um telefone fixo. Ele deve exibir o número de mensagens gravadas em um *display* de LED e deve permitir que o usuário disque e ouça as mensagens gravadas.

20.4 Explique por que uma abordagem orientada a objetos para desenvolvimento de software pode não ser adequada para sistemas de tempo real.

20.5 Mostre como o padrão Controle de Ambiente poderia ser usado como base para o projeto de um sistema para controlar a temperatura em uma estufa. A temperatura deve ser entre 10° C e 30° C. Se ela cai abaixo de 10° C, o sistema de aquecimento deve ser acionado; se ela vai acima de 30° C, uma janela deve ser aberta automaticamente.

20.6 Projete uma arquitetura de processo para um sistema de monitoração ambiental que coleta dados de um conjunto de sensores de qualidade de ar situados em torno de uma cidade. Existem cinco mil sensores organizados em cem bairros. Cada sensor deve ser interrogado quatro vezes por segundo. Quando mais de 30% dos sensores em determinado bairro indicam que a qualidade do ar está abaixo de um nível aceitável, luzes de aviso local são ativadas. Todos os sensores retornam as leituras para um computador central, que gera relatórios sobre a qualidade do ar na cidade a cada 15 minutos.

20.7 Um sistema de proteção de trem aciona automaticamente os freios se o limite de velocidade para um segmento de via for excedido ou se o trem entrar em um segmento de via que esteja sinalizado com uma luz vermelha (ou seja, o segmento não deve ser ocupado). Os detalhes são mostrados no Quadro 20.1. Identifique os estímulos que devem ser processados pelo sistema de controle de trem a bordo e as respostas associadas a esses estímulos.

20.8 Sugira uma possível arquitetura de processo para esse sistema.

20.9 Se um processo periódico do sistema de proteção de trem a bordo é usado para coletar dados do transmissor de via, com que frequência ele deve ser programado para assegurar que o sistema possa coletar as informações do transmissor? Explique como você chegou a sua resposta.

20.10 Por que os sistemas operacionais de uso geral, como Linux ou Windows, não são adequados para as plataformas de sistema de tempo real? Use sua experiência com sistemas de uso geral para responder a essa pergunta.

Quadro 20.1 Requisitos para um sistema de proteção de trem

Sistema de proteção de trem

- O sistema adquire informações sobre o limite de velocidade de um segmento de um transmissor de via que transmite continuamente o identificador do segmento e seu limite de velocidade. O mesmo transmissor também transmite informações sobre o *status* do sinal que está controlando esse segmento de via. O tempo necessário para transmitir informações de segmento e de sinal é de 50 ms.
- O trem pode receber informações do transmissor de via quando está a cerca de 10 m de um transmissor.
- A velocidade máxima do trem é 180 km/h.
- Os sensores no trem fornecem informações sobre sua atual velocidade (atualizadas a cada 250 ms) e o *status* do freio (atualizado a cada 100 ms).
- Se a velocidade do trem exceder o limite de velocidade de segmento atual por mais de 5 km/h, um aviso soa na cabine do condutor. Se a velocidade do trem exceder o limite de velocidade do segmento atual por mais de 10 km/h, os freios do trem são acionados automaticamente até que a velocidade caia para o limite de velocidade do segmento. Os freios do trem devem ser acionados dentro de 100 ms do momento em que a velocidade excessiva do trem foi detectada.
- Se o trem entrar em uma via sinalizada, ou seja, sinalizada com uma luz vermelha, o sistema de proteção de trem aciona os freios do trem e reduz a velocidade para zero. Os freios do trem devem ser acionados dentro de 100 ms do momento em que o sinal de luz vermelha é recebido.
- O sistema atualiza continuamente um *display* de *status* na cabine do maquinista.

REFERÊNCIAS

BERRY, G. Real-time programming: Special-purpose or general-purpose languages. In: RITTER, G. (Org.). *Information Processing*. Amsterdam: Elsevier Science Publishers, 1989, p. 11-17.

BRINCH-HANSEN, P. *Operating System Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

BURNS, A.; WELLINGS, A. *Real-Time Systems and Programming Languages*: Ada, Real-Time Java and C/Real-Time POSIX. Boston: Addison-Wesley, 2009.

COOLING, J. *Software Engineering for Real-Time Systems*. Harlow, Reino Unido: Addison-Wesley, 2003.

DIBBLE, P. C. *Real-time Java Platform Programming*. 2. ed. Charleston, SC: Booksurge Publishing, 2008.

DIJKSTRA, E. W. Cooperating Sequential Processes. In: GENUYS, F. (Org.). *Programming Languages*. Londres: Academic Press, 1968, p. 43-112.

DOUGLASS, B. P. *Real-Time UML*: Developing Efficient Objects for Embedded Systems. 2. ed. Boston: Addison-Wesley, 1999.

_______. *Real-Time Design Patterns*: Robust Scalable Architecture for Real-Time Systems. Boston: Addison-Wesley, 2002.

GOMAA, H. *Software Design Methods for Concurrent and Real-Time Systems*. Reading, Mass.: Addison-Wesley, 1993.

HAREL, D. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Programming*, v. 8, n. 3, 1987, p. 231--274.

_______. On Visual Formalisms. *Comm. ACM*, v. 31, n. 5, 1988, p. 514-530.

HOARE, C. A. R. Monitors: an operating system structuring concept. *Comm. ACM*, v. 21, n. 8, 1974, p. 666-677.

LEE, E. A. Embedded Software. In: ZELKOWITZ, M. (Org.). *Advances in Computers*. Londres: Academic Press, 2002.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating System Concepts*, 8. ed. Nova York: John Wiley & Sons 2008.

TANENBAUM, A. S. *Modern Operating Systems*. 3. ed. Englewood Cliffs, NJ: Prentice Hall, 2007.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 **21** 22 23 24 25 26

Engenharia de software orientada a aspectos

Objetivos

O objetivo deste capítulo é fazer uma introdução ao desenvolvimento de software orientado a aspectos, que é baseado na separação de interesses. Com a leitura deste capítulo, você:

- entenderá por que a separação de interesses é um bom princípio para desenvolvimento de software;
- terá sido introduzido às ideias fundamentais por trás de aspectos e desenvolvimento de software orientado a aspectos;
- entenderá como uma abordagem orientada a aspectos pode ser usada para engenharia de requisitos, projeto de software e programação;
- estará ciente das dificuldades para testar sistemas orientados a aspectos.

Conteúdo

Na maioria dos sistemas de grande porte, os relacionamentos entre requisitos e componentes de programa são complexos. Um único requisito pode ser implementado por uma série de componentes, e cada componente pode incluir elementos de vários requisitos. Na prática, isso significa que mudar requisitos pode envolver o entendimento e a alteração de vários componentes. Como alternativa, um componente pode prover alguma funcionalidade central, mas também incluir o código que implementa vários requisitos de sistema. Mesmo quando parece haver reúso significativo em potencial, pode ficar caro reusar tais componentes. O reúso pode envolver sua alteração para remover um código extra que não esteja associado com a funcionalidade central do componente.

Engenharia de software orientada a aspectos (AOSE, do inglês *aspect-oriented software engineering*) é uma abordagem para desenvolvimento de software que se propõe a resolver esse problema e tornar os programas mais fáceis de manter e reusar. AOSE é baseada em abstrações chamadas 'aspectos', que implementam funcionalidade de sistema que pode ser requerida em vários lugares diferentes em um programa. Os aspectos encapsulam a funcionalidade que cruza e coexiste com outra funcionalidade que existe no sistema. Eles são usados junto com outras abstrações, como objetos e métodos. Um programa executável orientado a aspectos é criado pela combinação (composição) automática de objetos, métodos e aspectos, de acordo com as especificações que são incluídas no código-fonte de programa.

Uma característica importante de aspectos é que eles incluem uma definição sobre onde devem ser incluídos em um programa, assim como o código que implementa o interesse transversal. Você pode especificar que o código transversal deve ser incluído antes ou depois da chamada de um método específico ou quando um atributo é acessado. Essencialmente, o aspecto é composto no programa para criar um novo sistema aumentado.

O principal benefício de uma abordagem orientada a aspectos é que ela suporta a separação de interesses. Como explico na Seção 21.1, separar interesses em elementos diferentes em vez de incluir interesses diferentes na mesma abstração lógica é uma boa prática de engenharia de software. Ao apresentar interesses transversais como aspectos, esses interesses podem ser entendidos, reusados e modificados de forma independente, sem a preocupação de onde o código é usado. Por exemplo, a autenticação de usuário pode ser representada como um aspecto que requisita um nome de usuário e uma senha. Isso pode ser automaticamente embutido no programa sempre que uma autenticação for requerida.

Digamos que você tenha um requisito de que a autenticação de usuário seja requerida antes de qualquer alteração dos detalhes pessoais ser feita no banco de dados. Você pode descrever isso como um aspecto, estabelecendo que o código de autenticação deve ser incluído antes de toda chamada a métodos que atualizam detalhes pessoais. Posteriormente, pode ampliar o requisito para a autenticação de todas as atualizações do banco de dados. Isso pode ser facilmente implementado com alteração do aspecto. Você pode simplesmente mudar a definição sobre onde o código de autenticação deve ser composto no sistema. Não precisa procurar no sistema todas as ocorrências desses métodos. Dessa forma, a chance de cometer erros e introduzir acidentalmente vulnerabilidades de proteção no programa é menor.

Pesquisa e desenvolvimento sobre orientação a aspectos têm como foco principal a programação orientada a aspectos. As linguagens de programação orientadas a aspectos como AspectJ (COLYER e CLEMENT, 2005; COLYER et al., 2005; KICZALES et al., 2001; LADDAD 2003a; LADDAD, 2003b) foram desenvolvidas de forma a ampliar a programação orientada a objetos para incluir aspectos. As principais empresas usaram programação orientada a aspectos em seus processos de produção de software (COLYER e CLEMENT, 2005). No entanto, os interesses transversais são igualmente problemáticos em outros estágios do processo de desenvolvimento de software. Os pesquisadores estão investigando, atualmente, como utilizar orientação a aspectos na engenharia de requisitos de sistema e na modelagem de sistema e como testar e verificar programas orientados a aspectos.

Eu incluí a discussão sobre AOSE aqui porque seu foco na separação de interesses é uma forma importante de se pensar sobre estruturar um sistema de software. Embora alguns sistemas de grande escala tenham implementado uma abordagem orientada a aspectos, o uso de aspectos ainda não faz parte da corrente principal da engenharia de software. Assim como acontece com todas as novas tecnologias, os defensores dão mais ênfase aos benefícios em vez de problemas e custos. Embora ainda leve algum tempo antes que a AOSE seja usada junto com outras abordagens da engenharia de software, a ideia de separação de interesses que fundamenta a AOSE é importante. Pensar na separação de interesses é uma boa abordagem geral para engenharia de software.

Portanto, nas seções restantes deste capítulo, focalizarei os conceitos que são parte de AOSE e discutirei as vantagens e as desvantagens do uso de uma abordagem orientada a aspectos em estágios diferentes do processo de desenvolvimento de software. Como meu objetivo é ajudar você a entender os conceitos que estão por trás da AOSE, não entro em detalhes de qualquer abordagem ou linguagem específica de programação orientada a aspectos.

21.1 Separação de interesses

A separação de interesses é um princípio-chave de projeto e implementação de software. Isso significa que você deve organizar seu software de tal forma que cada elemento do programa (classe, método, procedimento etc.) faça apenas uma coisa. Assim poderá concentrar-se nesse elemento sem se preocupar com outros elementos no programa. Poderá entender cada parte do programa conhecendo seus interesses, sem a necessidade de entender outros elementos. Quando as mudanças forem necessárias, elas serão feitas em um número pequeno de elementos.

A importância de separação de interesses foi reconhecida nos primeiros estágios da história da ciência da computação. Subrotinas, que encapsulam uma unidade de funcionalidade, foram inventadas no início da década de 1950 e mecanismos de estruturação de programas posteriores, como procedimentos e classes de objeto, foram projetados para prover mecanismos melhores para realizar a separação de interesses. No entanto, todos esses mecanismos possuem problemas em lidar com certos tipos de interesses que cruzam outros interesses. Esses interesses transversais não podem ser localizados por meio de mecanismos estruturados como objetos e funções. Aspectos foram inventados para ajudar a gerenciar esses interesses transversais.

Embora seja aceito de forma generalizada que a separação de interesses é uma boa prática de engenharia de software, é mais difícil definir o que de fato significa um interesse nesse contexto. Às vezes, é definido como uma noção funcional (por exemplo, um interesse é algum elemento ou funcionalidade em um sistema). Como alterna-

tiva, ele pode ser definido muito amplamente como 'qualquer parte de interesse ou foco de um programa'. Nenhuma dessas definições é muito útil na prática; os interesses são certamente mais que simples elementos funcionais, porém a definição mais generalizada é tão vaga que é praticamente inútil.

Sob meu ponto de vista, a maioria das tentativas de definir interesses é problemática porque tenta relacionar os interesses com os programas. De fato, conforme discutido por Jacobson e Ng (2004), os interesses são realmente reflexos dos requisitos de sistema e prioridades dos *stakeholders* no sistema. O desempenho de sistema pode ser um interesse porque os usuários querem uma resposta rápida do sistema; alguns *stakeholders* podem estar interessados em que o sistema inclua determinada funcionalidade; pode ser interessante a empresas que dão suporte ao sistema que o sistema seja fácil de manter. Dessa forma, um interesse pode ser definido como algo que é importante ou significativo para um *stakeholder* ou um grupo de *stakeholders*.

Se você pensar em interesses como uma maneira de organizar requisitos, pode perceber por que uma abordagem para implementação que separa os interesses em diferentes elementos de programa é uma boa prática. É mais fácil rastrear interesses expressos na forma de requisitos ou conjuntos de requisitos relacionados para componentes de programa que implementam esses interesses. Se os requisitos mudarem, então a parte do programa que deve ser alterada fica evidente.

Existem vários tipos diferentes de interesses de *stakeholder*:

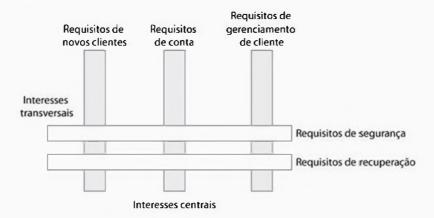
1. Interesses funcionais, relacionados com uma funcionalidade específica a ser incluída no sistema. Por exemplo, em um sistema de controle de trens, um interesse funcional específico seria a frenagem do trem.
2. Interesses de qualidade de serviço, relacionados ao comportamento não funcional de um sistema. Incluem características como desempenho, confiabilidade e disponibilidade.
3. Interesses de políticas, relacionados com as políticas gerais que governam o uso de um sistema. Interesses de políticas incluem interesses de segurança e interesses relacionados com regras de negócio.
4. Interesses de sistemas, relacionados com atributos do sistema como um todo, tais como manutenibilidade e configurabilidade.
5. Interesses organizacionais, relacionados com objetivos e prioridades organizacionais. Incluem a produção de um sistema dentro do orçamento, fazendo uso de ativos existentes de software e mantendo a reputação da organização.

Os interesses centrais de um sistema são aqueles interesses funcionais que se relacionam com seu propósito principal. Portanto, para um sistema hospitalar de informações de pacientes, por exemplo, os interesses funcionais centrais são a criação, a edição, a consulta e o gerenciamento de registros de pacientes. Além dos interesses centrais, os grandes sistemas possuem também interesses funcionais secundários. Estes podem envolver funcionalidade que compartilha informações com os interesses centrais, ou que é requerido para que o sistema possa satisfazer seus requisitos não funcionais.

Por exemplo, considere um sistema que tenha um requisito de prover acesso concorrente a um *buffer* compartilhado. Um processo adiciona dados no *buffer*, e outro processo obtém dados a partir do mesmo *buffer*. Esse *buffer* compartilhado é parte de um sistema de aquisição de dados em que um processo produtor coloca dados no *buffer* e um processo consumidor obtém os dados do *buffer*. O interesse central aqui é manter o *buffer* compartilhado, então a funcionalidade central é associada com adição e remoção de elementos do *buffer*. No entanto, para garantir que os processos produtor e consumidor não interfiram entre si, existe um interesse secundário essencial de sincronização. O sistema deve ser projetado de tal forma que o processo produtor não possa sobrescrever os dados que não foram consumidos e que o processo consumidor não possa ler os dados de um *buffer* vazio.

Além desses interesses secundários, outros interesses como qualidade de serviço e políticas organizacionais refletem requisitos essenciais de sistema. Em geral, esses são os interesses de sistema — eles se aplicam ao sistema como um todo, em vez de requisitos individuais ou realização desses requisitos de um programa. Eles são chamados interesses transversais, para que sejam diferenciados dos interesses centrais. Os interesses funcionais secundários também podem ser transversais, embora nem sempre cruzem o sistema inteiro. Em vez disso, eles são associados a agrupamentos de interesses centrais que provêm a funcionalidade relacionada.

Interesses transversais são mostrados na Figura 21.1, baseada em um exemplo de um sistema de Internet banking. Esse sistema possui requisitos relacionados novos clientes, como verificação de crédito e endereço. Ele também tem requisitos relacionados com o gerenciamento de clientes existentes e gerenciamento de contas de clientes. Todos esses são interesses centrais que estão associados com o objetivo principal do sistema — fornecimento de um serviço de Internet banking. No entanto, o sistema possui também requisitos de segurança baseados em políticas de proteção do banco e requisitos de recuperação para garantir que os dados não sejam perdidos

Figura 21.1 Interesses transversais

no caso de uma falha de sistema. Esses são os interesses transversais, porque podem influenciar a implementação de todos os outros requisitos de sistema.

Abstrações de linguagem de programação, como procedimentos e classes, são os mecanismos que normalmente usamos para organizar e estruturar os interesses centrais de um sistema. No entanto, a implementação dos interesses centrais nas linguagens de programação convencionais normalmente inclui código adicional para implementar interesses transversais, funcionais, de qualidade de serviço e de políticas. Isso gera dois fenômenos indesejáveis: entrelaçamento (*tangling*) e espalhamento (*scattering*).

O entrelaçamento ocorre quando um módulo do sistema inclui código que implementa diferentes requisitos. O exemplo no Quadro 21.1, que é uma implementação simplificada da parte do código para um sistema de delimitação de *buffer*, ilustra esse fenômeno. O quadro é uma implementação da operação *put* que adiciona um item no *buffer*. No entanto, se o *buffer* estiver cheio, é necessário aguardar até que a operação *get* correspondente remova um item do *buffer*. Os detalhes não são importantes; essencialmente, as chamadas wait() e notify() são usadas para sincronizar as operações *put* e *get*. O código que apoia o interesse central (nesse caso, inserir um registro no *buffer*) está entrelaçado com o código que implementa a sincronização. O código de

Quadro 21.1 Entrelaçamento do código de gerenciamento e sincronização de *buffer*

```
synchronized void put (SensorRecord rec )
{
    // Verifica se há espaço no buffer; espera se não houver
    if ( numberOfEntries == bufsize)
        wait () ;
    // Adiciona registro no fim do buffer
    store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
    back = back + 1 ;
    // Se está no fim do buffer, a próxima entrada está no começo
    if (back == bufsize)
        back = 0 ;
    numberOfEntries = numberOfEntries + 1 ;
    // Indica que buffer está disponível
    notify () ;
} // put
```

sincronização, que está associado ao interesse secundário de garantir a exclusão mútua, deve ser incluído em todos os métodos que acessam o *buffer* compartilhado. O código associado com o interesse de sincronização está sombreado na Figura 21.2.

O fenômeno relacionado de espalhamento ocorre quando a implementação de um único interesse (um requisito lógico ou conjunto de requisitos) está espalhada por vários componentes em um programa. É mais provável que isso ocorra quando os requisitos relacionados com os interesses funcionais secundários ou interesses de políticas são implementados.

Por exemplo, imagine um sistema de gerenciamento de registros médicos, como MHC-PMS, que tem uma série de componentes cujo interesse está em gerenciar as informações pessoais, medicações, consultas, imagens médicas, diagnósticos e tratamentos. Esses componentes implementam o interesse central do sistema: manter registros dos pacientes. O sistema pode ser configurado para diferentes tipos de clínicas ao selecionar os componentes que fornecem a funcionalidade requerida pela clínica.

No entanto, suponha que exista também um interesse secundário importante — a manutenção de informações estatísticas. O provedor de código de saúde deseja que sejam gravados detalhes sobre quantos pacientes foram admitidos e dispensados todo mês, quantos pacientes faleceram, que medicações foram receitadas, os motivos das consultas e assim por diante. Esses requisitos devem ser implementados com adição de código, que torna os dados anônimos (para manter a privacidade do paciente) e grava esses dados em um banco de dados de estatísticas. Um componente estatístico processa os dados estatísticos e gera relatórios necessários.

Isso está na Figura 21.2. O diagrama mostra exemplos de três classes que poderiam ser incluídas no sistema de registro de pacientes junto com alguns métodos centrais para gerenciar informações de pacientes. A área sombreada mostra os métodos necessários para implementar o interesse estatístico secundário. Você pode observar que o interesse estatístico está espalhado por outros interesses centrais.

Os problemas com entrelaçamento e espalhamento ocorrem quando os requisitos iniciais de sistema mudam. Por exemplo, digamos que novos dados estatísticos precisam ser colhidos no sistema de registro de pacientes. As alterações do sistema não são todas feitas em um único lugar e, por isso, você precisa gastar tempo procurando os componentes do sistema que devem ser alterados. Depois, terá de alterar cada um desses componentes para incorporar as mudanças necessárias. Isso pode ser caro por causa do tempo necessário para analisar os componentes e, depois, para fazer e testar as alterações. Sempre existe a possibilidade de você esquecer algum código que deveria ser alterado. Dessa forma, as estatísticas estariam incorretas. Além disso, como várias alterações devem ser feitas, isso aumenta as chances de erros e defeitos no sistema.

21.2 Aspectos, pontos de junção e pontos de corte

Nesta seção, introduzo os mais importantes novos conceitos associados com o desenvolvimento de software orientado a aspectos e os ilustro usando exemplos de MHC-PMS. A terminologia que uso foi introduzida pelos desenvolvedores de AspectJ no fim da década de 1990. No entanto, os conceitos são aplicáveis de modo geral e não são específicos para a linguagem de programação AspectJ. A Tabela 21.1 resume os principais termos que você precisa entender.

Figura 21.2 Espalhamento de métodos que implementam interesses secundários

Paciente	Imagem	Consulta
<attribute decls>	<attribute decls>	<attribute decls>
getName () editName () getAddress () editAddress () ...	getModality () archive () getDate () editDate () ...	makeAppoint () cancelAppoint () assignNurse () bookEquip () ...
anonymize () ...	saveDiagnosis () saveType () ...	anonymize () saveConsult () ...

Tabela 21.1 Terminologia usada na engenharia de software orientada a aspectos

Termo	Definição
adendo	Código que implementa um interesse.
aspecto	Uma abstração de programa que define o interesse transversal. Inclui a definição de um ponto de corte e do adendo associado com esse interesse.
ponto de junção	Evento em um programa em execução onde o adendo associado com um aspecto pode ser executado.
modelo de ponto de junção	Conjunto de eventos que podem ser referenciados em um ponto de corte.
ponto de corte	Uma declaração, inclusa em um aspecto, que define os pontos de junção onde o adendo de aspecto associado deve ser executado.
composição	A incorporação do código de adendo em ponto de junção específico por um compositor de aspectos.

Um sistema de registros médicos como MHC-PMS inclui componentes que lidam com informações de pacientes relacionadas logicamente. O componente paciente mantém a informação pessoal sobre um paciente, o componente medicação guarda a informação sobre os remédios receitados, e assim por diante. Ao projetar o sistema usando a abordagem baseada em componentes, instâncias diferentes do sistema podem ser configuradas. Por exemplo, uma versão poderia ser configurada para cada tipo de clínica onde os médicos pudessem receitar a medicação relevante apenas a essa clínica. Isso simplifica o trabalho do pessoal da clínica e reduz as chances de um médico receitar a medicação errada.

No entanto, essa organização significa que a informação no banco de dados deve ser atualizada a partir de vários lugares do sistema. Por exemplo, a informação de um paciente pode ser alterada quando seus detalhes pessoais mudam, quando a medicação receitada muda, quando ele é encaminhado a um novo especialista etc. Para simplificar, imagine que todos os componentes do sistema usem uma estratégia de nomes consistente e que todas as atualizações do banco de dados sejam implementadas em métodos que iniciam com a palavra 'update'. Dessa forma, haverá no sistema métodos como:

```
updatePersonalInformation (patientId, infoupdate)

updateMedication (patientId, medicationupdate)
```

O paciente é identificado através de patientID, e as mudanças a serem feitas estão codificadas dentro do segundo parâmetro; os detalhes dessa codificação não são importantes para esse exemplo. As atualizações são feitas pelos funcionários do hospital, que estão autenticados pelo sistema.

Imagine que ocorra uma brecha de proteção onde a informação do paciente seja alterada de forma maliciosa. Talvez alguém tenha deixado acidentalmente o computador autenticado no sistema e uma pessoa não autorizada tenha obtido acesso. Ou, então, uma pessoa autorizada pode ter obtido um acesso maior e alterado maliciosamente a informação de paciente. Para reduzir a possibilidade de isso acontecer novamente, uma nova política de proteção é definida. Antes de ser efetuada qualquer alteração no banco de dados de pacientes, a pessoa que está requisitando a alteração deve autenticar-se novamente no sistema. Os detalhes de quem fez a alteração são gravados em um arquivo separado. Isso ajudará a rastrear os problemas, caso ocorram novamente.

Uma forma de implementar essa nova política é modificar o método *update* em cada componente para chamar outros métodos que façam autenticação e registro. Como alternativa, o sistema poderia ser alterado de tal forma que, cada vez que um método *update* for chamado, as chamadas de método sejam adicionadas antes de a chamada fazer a autenticação e depois de uma chamada registrar as mudanças feitas. No entanto, nenhuma das duas é uma boa solução para o problema:

1. A primeira abordagem gera uma implementação entrelaçada. Logicamente, atualizar o banco de dados, autenticar quem faz a alteração e registrar os detalhes da atualização são interesses separados e não relacionados. Você pode querer incluir a autenticação em outros lugares do sistema sem registro ou pode querer registrar as ações independentemente da atualização. O mesmo código de autenticação e registro tem de ser incluído dentro de vários métodos diferentes.

2. A abordagem alternativa leva a uma implementação espalhada. Se você incluir explicitamente as chamadas aos métodos para fazer a autenticação e o registro antes e depois de cada chamada aos métodos *update*, então esse código será incluído em vários lugares diferentes do sistema.

Autenticação e registro atravessam os interesses centrais do sistema e podem ter de ser incluídos em vários lugares. Em um sistema orientado a aspectos, você pode representar esses interesses transversais como aspectos separados. Um aspecto inclui uma especificação sobre onde o interesse transversal deve ser inserido no programa e o código para implementá-lo. Isso é demonstrado no Quadro 21.2, que define um aspecto de autenticação. A notação que uso no exemplo segue o estilo de AspectJ, porém, com uma sintaxe simplificada, que deve ser compreensível sem o conhecimento de Java ou AspectJ.

Aspectos são totalmente diferentes de outras abstrações de programa, pois o aspecto em si inclui uma especificação sobre onde ele deve ser executado. Em outras abstrações, como métodos, existe uma separação clara entre a definição da abstração e de seu uso. Você não pode dizer apenas ao analisar um método a partir de onde ele será chamado; as chamadas podem vir de qualquer lugar de onde o método seja visível. Ao contrário disso, os aspectos incluem um 'ponto de corte' — uma declaração que define onde o aspecto será composto no programa.

Neste exemplo, o ponto de corte é uma declaração simples:

```
before: call (public void update* (..))
```

O significado disso é que, antes da execução de qualquer método cujo nome inicie com a palavra 'update', seguida por qualquer outra sequência de caracteres, o código no aspecto após a definição do ponto de corte deve ser executado. O caractere * é chamado de coringa e substitui quaisquer caracteres que sejam permitidos em identificadores. O código a ser executado é chamado 'adendo' e é a implementação do interesse transversal. Nesse caso, o adendo obtém a senha da pessoa que está requisitando a alteração e verifica se é igual à senha do usuário autenticado atualmente. Se não for igual, é feito o *logoff* do usuário e a atualização não prossegue.

Quadro 21.2 Um aspecto de autenticação

```
aspect authentication
{
   before: call (public void update* (..)) // este é o ponto de corte
   {
      // este é o adendo que deve ser executado quando inserido no //sistema em execução
      int tries = 0 ;
      string userPassword = Password.Get ( tries ) ;
      while (tries < 3 && userPassword != thisUser.password ( ) )
      {
          // permite três tentativas de senha correta
          tries = tries + 1 ;
          userPassword = Password.Get ( tries ) ;
      }
      if (userPassword != thisUser.password ( )) then
          // se a senha estiver errada, assume que usuário tenha esquecido de fazer logoff
          System.Logout (thisUser.uid) ;
      }
} // authentication
```

A capacidade de especificar, com pontos de corte, onde o código deve ser executado, é a característica que diferencia os aspectos. No entanto, para entender o que significa ponto de corte, você precisa entender outro conceito — o ponto de junção. Um ponto de junção é um evento que ocorre durante a execução de um programa. Pode ser uma chamada de método, a iniciação de uma variável, atualização de um campo etc.

Existem vários tipos possíveis de eventos que podem ocorrer durante a execução de um programa. Um modelo de ponto de junção define o conjunto de eventos que podem ser referenciados em um programa orientado a aspectos. Modelos de pontos de junção não são padronizados, e cada linguagem de programação orientada a aspectos possui seu próprio modelo de pontos de junção. Por exemplo, em AspectJ, os eventos que fazem parte do modelo de pontos de junção incluem:

- *Eventos de chamada* — chama um método ou um construtor;
- *Eventos de execução* — a execução de um método ou de um construtor;
- *Eventos de iniciação* — iniciação de classe ou objeto;
- *Eventos de dados* — acesso ou atualização de um campo;
- *Eventos de exceção* — tratamento de uma exceção.

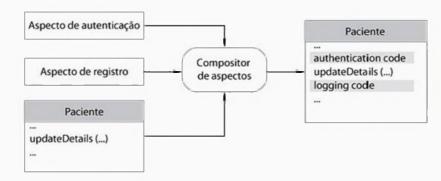
Um ponto de corte identifica evento(s) específico(s) — por exemplo, uma chamada a um procedimento identificado — com o(s) qual(is) o adendo deve ser associado. Isso significa que você pode compor o adendo em um programa em vários contextos diferentes, dependendo do modelo suportado de pontos de junção:

1. Adendo pode ser inserido antes da execução de um método específico, uma lista de métodos identificados ou uma lista de métodos cujos nomes correspondem a um padrão especificado (como *update**).
2. Adendo pode ser inserido depois do retorno normal ou excepcional de um método. No exemplo do Quadro 21.2, você poderia definir um ponto de corte que executaria o código de registro depois de todas as chamadas a métodos *update*.
3. Adendo pode ser inserido quando um campo de um objeto é modificado; você pode incluir um adendo para monitorar ou alterar esse campo.

A inclusão do adendo em pontos de junção especificados nos pontos de corte é responsabilidade de um compositor de aspectos. Os compositores de aspectos são extensões dos compiladores que processam a definição de aspectos e classes de objetos e métodos que definem o sistema. O compositor então gera um novo programa com aspectos inclusos em pontos de junção específicos. Os aspectos são integrados de forma que os interesses transversais sejam executados em locais corretos no sistema final.

A Figura 21.3 ilustra essa composição de aspectos por aspectos de autenticação e registro que devem ser incluídos em MHC-PMS. Existem três abordagens diferentes para composição de aspectos:

1. Pré-processamento de código-fonte, em que um compositor recebe um código-fonte e gera um novo código--fonte em uma linguagem como Java ou C++, que pode, então, ser compilada usando o compilador-padrão de linguagem. Essa abordagem foi adotada para linguagem AspectX com seu associado XWeaver (BIRRER et al., 2005).
2. Composição em tempo de ligação, em que o compilador é modificado para incluir um compositor de aspectos. Uma linguagem orientada a aspectos como AspectJ é processada e um *bytecode*-padrão Java é gerado. Este pode, então, ser executado diretamente por um interpretador Java ou processado novamente para gerar código de máquina nativo.

Figura 21.3 Composição de aspectos

3. Composição dinâmica em tempo de execução. Nesse caso, pontos de junção são monitorados, e, quando um evento referenciado em um ponto de corte ocorre, o adendo correspondente é integrado com o programa em execução.

A abordagem mais comum para a composição de aspectos é a composição em tempo de ligação, porque isso permite uma implementação eficiente de aspectos sem um grande *overhead* em tempo de execução. Composição dinâmica é a abordagem mais flexível, mas pode afetar seriamente o desempenho durante a execução do programa. Pré-processamento de código-fonte é raramente usado hoje em dia.

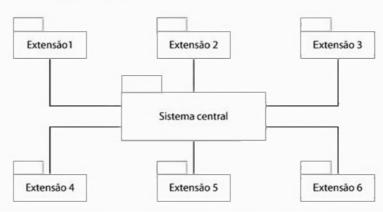
21.3 Engenharia de software com aspectos

Aspectos foram introduzidos originalmente como uma construção de linguagem de programação, mas, conforme vimos, a noção de interesses vem de requisitos de sistema. Portanto, faz sentido adotar uma abordagem orientada a aspectos em todos os estágios do processo de desenvolvimento de software. Nos estágios iniciais da engenharia de software, adotar uma abordagem orientada a aspectos significa usar a ideia de separação de interesses como base para pensar nos requisitos e projeto de sistema. Identificar e modelar interesses deve fazer parte dos processos de engenharia de software e de modelagem. Linguagens de programação orientadas a aspectos fornecem, então, o suporte tecnológico para manter a separação de interesses em sua implementação do sistema.

Jacobson e Ng (2004) sugerem que, ao projetar um sistema, você pense sobre o sistema suportando interesses de diferentes *stakeholders* como sendo um sistema central com extensões. Isso está ilustrado na Figura 21.4, na qual usei pacotes UML para representar a parte central e as extensões. O sistema central é um conjunto de recursos do sistema que implementa seu propósito essencial. Portanto, se o propósito de determinado sistema é manter as informações dos pacientes de um hospital, o sistema central fornece meios para criar, editar, gerenciar e acessar um banco de dados de registros de pacientes. As extensões do sistema central refletem os interesses adicionais, que devem ser integrados com o sistema central. Por exemplo, é importante que um sistema de informações médicas mantenha a confidencialidade das informações de pacientes. Dessa forma, uma extensão pode preocupar-se com controle de acesso, outra com criptografia etc.

Existem diversos tipos de extensões que derivam dos diferentes tipos de interesses discutidos na Seção 21.1:

1. *Extensões funcionais secundárias.* Adicionam novas capacidades à funcionalidade fornecida no sistema central. Por exemplo, baseando-se no sistema MHC-PMS, a produção de relatórios de remédios receitados no mês anterior seria uma extensão funcional secundária de um sistema de informação de pacientes.
2. *Extensões de políticas.* Adicionam capacidades funcionais para suportar políticas organizacionais. Extensões que acrescentam recursos de proteção são exemplos de extensões de políticas.
3. *Extensões de qualidade de serviço* (*QoS*, do inglês *Quality of Service*). Adicionam capacidades funcionais para ajudar a alcançar a qualidade de requisitos do serviço que foram especificados para o sistema. Por exemplo, uma extensão pode implementar um *cache* para reduzir o número de acessos ao banco de dados ou *backups* automatizados para recuperação em caso de falha de sistema.

Figura 21.4 Sistema central com extensões

4. *Extensões de infraestrutura.* Adicionam capacidades funcionais para suportar a implementação de um sistema em algumas plataformas específicas. Por exemplo, em um sistema de informação de pacientes, as extensões de infraestrutura poderiam ser usadas para implementar a interface para o sistema de gerenciamento de banco de dados usado subjacente. As alterações nessa interface podem ser feitas modificando-se as extensões de infraestrutura associadas.

As extensões sempre acrescentam algum tipo de funcionalidade ou recursos adicionais para o sistema central. Os aspectos são uma maneira para implementar essas extensões e podem ser compostos com a funcionalidade do sistema central usando recursos de composição do ambiente de programação orientado a aspectos.

21.3.1 Engenharia de requisitos orientada a interesses

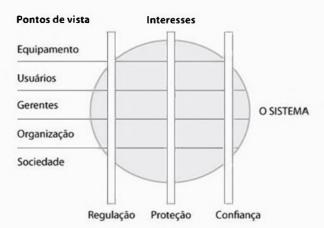
Conforme sugerido na Seção 21.1, os interesses refletem os requisitos dos *stakeholders*. Esses interesses podem refletir a funcionalidade requerida por um *stakeholder*, a qualidade de serviço do sistema, políticas organizacionais ou questões relacionadas aos atributos do sistema como um todo. Por isso, faz sentido adotar uma abordagem para engenharia de requisitos que identifique e especifique os interesses dos diferentes *stakeholders*. O termo 'aspectos iniciais' é usado, às vezes, em referência ao uso de aspectos nos estágios iniciais do ciclo de vida do software onde a separação de interesses é enfatizada.

A importância de separar os interesses durante a engenharia de requisitos tem sido reconhecida há muitos anos. Pontos de vista que representam diferentes perspectivas de sistema foram incorporados em uma série de métodos da engenharia de requisitos (EASTERBROOK e NUSEIBEH, 1969; FINKELSTEIN et al., 1992; KOTONYA e SOMMERVILE, 1996). Esses métodos separam os interesses dos diferentes *stakeholders*. Os pontos de vista refletem a funcionalidade distinta requerida por diferentes grupos de *stakeholders*.

No entanto, existem também requisitos que cruzam todos os pontos de vista, conforme mostrado na Figura 21.5. Esse diagrama mostra que os pontos de vista podem ser de tipos diferentes, porém os interesses transversais (como regulação, confiança e proteção) geram requisitos capazes de impactar em todos os pontos de vista. Essa foi a maior consideração no trabalho que fiz durante o desenvolvimento do método PreView (SOMMERVILLE e SAWYER, 1997; SOMMERVILLE et al., 1998), que inclui passos para identificar interesses transversais não funcionais.

Para desenvolver um sistema organizado no estilo da Figura 21.4, você deve identificar os requisitos para o sistema central junto com os requisitos das extensões de sistema. Uma abordagem orientada a pontos de vista para engenharia de requisitos, em que cada ponto de vista representa os requisitos de grupos relacionados de *stakeholders*, é uma maneira de separar interesses principais de secundários. Se você organizar os requisitos de acordo com os pontos de vista de *stakeholders*, poderá analisá-los para descobrir requisitos relacionados que aparecem em todos ou na maioria dos requisitos. Eles representam a funcionalidade central do sistema. Outros requisitos de ponto de vista podem ser requisitos que são específicos para esse ponto de vista e podem ser implementados como extensões da funcionalidade central.

Por exemplo, imagine que você está desenvolvendo um sistema de software para manter registro de equipamentos especializados usados por serviços emergenciais. Os equipamentos são localizados em lugares diferentes

Figura 21.5 Pontos de vista e interesses

de uma região ou estado e, em caso de uma emergência como enchente ou terremoto, os serviços emergenciais usam o sistema para descobrir qual equipamento está disponível próximo ao lugar do problema. O Quadro 21.3 mostra um esboço de requisitos de três pontos de vista possíveis para tal sistema.

Podemos observar a partir desse exemplo que os *stakeholders* de todos os pontos de vista diferentes precisam ser capazes de encontrar os equipamentos específicos, ver o equipamento disponível em cada localidade e emprestar/devolver o equipamento. Portanto, esses são os requisitos para o sistema central. Os requisitos secundários apoiam as necessidades mais específicas de cada ponto de vista. Existem requisitos secundários para as extensões do sistema suportando uso, gerenciamento e manutenção de equipamentos.

Os requisitos funcionais secundários identificados a partir de qualquer ponto de vista não necessariamente cruzam os requisitos dos outros pontos de vista. Por exemplo, apenas o ponto de vista de manutenção está interessado em completar os registros de manutenção. Esses requisitos refletem a necessidade desse ponto de vista e esses interesses podem não ser compartilhados com outros pontos de vista. No entanto, além dos requisitos funcionais secundários, existem interesses transversais que geram requisitos importantes para alguns ou todos os pontos de vista. Estes normalmente refletem os requisitos de políticas e de qualidade de serviço que se aplicam ao sistema como um todo. Conforme discutido no Capítulo 4, esses são requisitos não funcionais como os requisitos para proteção, desempenho e custo.

No sistema de inventário de equipamentos, um exemplo de um interesse transversal é a disponibilidade de sistema. As emergências podem ocorrer com pouco ou nenhum aviso. Salvar vidas pode requerer que o equipamento essencial seja disponibilizado o mais rapidamente possível. Portanto, os requisitos de confiança para o sistema de inventário de equipamentos incluem os requisitos para um alto nível de disponibilidade de sistema. Alguns exemplos desses requisitos de confiança, com a lógica associada, são mostrados no Quadro 21.4. Usando esses requisitos, você pode identificar as extensões de funcionalidade central para geração de relatórios de transações e de *status*. Essas extensões tornam mais fácil identificar problemas e comutar para um sistema de *backup*.

Quadro 21.3 Pontos de vista sobre um sistema de inventário de equipamentos

1. **Usuários de serviço de emergência**
   - 1.1 Encontrar um tipo específico de equipamento (por exemplo, tratores pesados)
   - 1.2 Ver equipamento disponível em um depósito específico
   - 1.3 *Check-out* de equipamento
   - 1.4 *Check-in* de equipamento
   - 1.5 Arranjar equipamento para ser transportado até o local da emergência
   - 1.6 Submeter relatório de danos
   - 1.7 Encontrar depósito próximo ao local da emergência
2. **Planejadores da emergência**
   - 2.1 Encontrar um tipo específico de equipamento
   - 2.2 Ver equipamento disponível em um depósito específico
   - 2.3 *Check-in/check-out* de equipamento para um depósito
   - 2.4 Mover equipamento de um depósito para outro
   - 2.5 Encomendar novo equipamento
3. **Pessoal de manutenção**
   - 3.1 *Check-in/check-out* de equipamento para um depósito para manutenção
   - 3.2 Ver equipamentos disponíveis em cada depósito
   - 3.3 Encontrar um tipo específico de equipamento
   - 3.4 Ver a agenda de manutenção para um item de equipamento
   - 3.5 Completar o registro de manutenção para um item de equipamento
   - 3.6 Mostrar todos os itens de um depósito que precisam de manutenção

Quadro 21.4 Requisitos relacionados com disponibilidade (AV, do inglês *availability*) para o sistema de inventário de equipamentos

AV.1 Deve existir um sistema '*hot stand-by*' disponível em uma localização bem separada geograficamente do sistema principal.
Razão: A emergência pode afetar a localização principal do sistema.
AV.1.1 Todas as transações devem ser registradas no local do sistema principal e no sistema *hot stand-by* remoto.
Razão: Isso permite que essas transações sejam repetidas e que o banco de dados do sistema esteja consistente.
AV.1.2 O sistema deve enviar informações de *status* para a sala de controle de emergência a cada cinco minutos.
Razão: Os operadores da sala de controle de emergência podem comutar para o sistema *hot stand-by* se o sistema principal estiver indisponível.

A saída do processo da engenharia de requisitos deve ser um conjunto de requisitos estruturados em torno da ideia de um sistema central com as extensões. Por exemplo, no sistema de inventário, exemplos de requisitos centrais poderiam ser:

C.1 O sistema permitirá que usuários autorizados vejam a descrição de qualquer item de equipamento no inventário de serviços de emergência.

C.2 O sistema deve incluir um recurso de busca para permitir que usuários autorizados busquem inventários individuais ou completos para um item específico de equipamento ou um tipo específico de equipamento.

O sistema também pode incluir uma extensão cujo propósito seja apoiar a aquisição e a troca de equipamentos. Os requisitos para essa extensão poderiam ser:

E1.1 Deve ser possível aos usuários autorizados informarem solicitações de substituição de itens de equipamento para fornecedores certificados.

E1.1.1 Quando um item de equipamento é solicitado, ele deve ser alocado para um inventário e marcado ali como 'solicitado'.

Como regra geral, você deve evitar ter interesses ou extensões demais no sistema. Isso simplesmente confunde o leitor e pode gerar um projeto prematuro. Isso limita a liberdade dos projetistas e pode resultar em um projeto de sistema incapaz de atender a seus requisitos de qualidade de serviço.

21.3.2 Projeto e programação orientados a aspectos

Projeto orientado a aspectos é o processo de projetar um sistema usando aspectos para implementar os interesses transversais e extensões que são identificados durante o processo de engenharia de software. Nesse estágio, você precisa traduzir os interesses relacionados com o problema a ser resolvido para aspectos correspondentes no programa que está implementando a solução. Você também deve compreender como esses aspectos serão compostos com outros componentes de sistema e garantir que não apareçam ambiguidades de composição.

A declaração de alto nível de requisitos fornece uma base para a identificação de algumas extensões de sistema que podem ser implementadas como aspectos. Depois, precisa desenvolvê-los em mais detalhes a fim de identificar outras extensões e entender a funcionalidade requerida. Uma forma de fazer isso é identificar um conjunto de casos de uso (assunto abordado nos capítulos 4 e 5) associado com cada ponto de vista. Os modelos de casos de uso são focados em interações e mais detalhados do que os requisitos de usuário. Pode-se pensar a seu respeito como uma ponte entre os requisitos e o projeto. Em um modelo de caso de uso, você descreve os passos de cada interação do usuário e, assim, começa a identificar e definir as classes do sistema.

Jacobson e Ng (2004) escreveram o livro que discute como os casos de uso podem ser usados na engenharia de software orientada a aspectos. Eles sugerem que cada caso de uso represente um aspecto e propõem extensões para abordagem de casos de uso para suportar pontos de junção e de corte. Eles introduzem, também, a ideia de fatias de casos de uso e módulos de casos de uso. Estes incluem fragmentos de classes que implementam um aspecto. Eles podem ser compostos para criar o sistema completo.

A Figura 21.6 mostra exemplos de três casos de uso que poderiam ser parte do sistema de gerenciamento de inventário. Eles refletem os interesses de adicionar um equipamento a um inventário e solicitar equipamento. Solicitar equipamento e adicionar equipamento a um depósito são interesses relacionados. Uma vez que os itens solicitados são entregues, eles devem ser adicionados ao inventário e entregues a um depósito de equipamentos.

Figura 21.6 Casos de uso do sistema de gerenciamento de inventário

A UML já inclui a noção de casos de uso de extensão. Um caso de uso de extensão estende a funcionalidade de outro caso de uso. A Figura 21.7 mostra como a criação de uma solicitação de equipamento estende o caso de uso central para adicionar equipamento a um depósito específico. Se o equipamento a ser adicionado não existe, ele pode ser solicitado e adicionado ao depósito quando for entregue. Durante o desenvolvimento de modelos de caso de uso, você deve procurar recursos comuns e, quando possível, estruturar casos de uso como casos centrais mais extensões. Recursos transversais, como registros de todas as transações, também podem ser representados como casos de uso de extensão. Jacobson e Ng discutem como extensões desse tipo podem ser implementadas como aspectos.

Desenvolver um processo efetivo para projeto orientado a aspectos é essencial se o projeto orientado a aspectos for aceito e usado. Eu sugiro que um processo de projeto orientado a aspectos deve incluir as atividades mostradas na Figura 21.8. Essas atividades são:

1. *Projeto de sistema central.* Nesse estágio, você projeta a arquitetura de sistema para suportar a funcionalidade central do sistema. A arquitetura deve levar em conta, também, requisitos de qualidade de serviço como requisitos de desempenho e confiança.
2. *Identificação e projeto de aspectos.* Começando com as extensões identificadas nos requisitos de sistema, você deve analisá-las para ver se são aspectos por si ou se deve quebrá-las em vários aspectos. Uma vez identificados os aspectos, eles podem ser projetados separadamente, levando-se em conta o projeto de recursos centrais de sistema.
3. *Projeto de composição.* Nesse estágio, você analisa o sistema central e projetos de aspectos para descobrir onde os aspectos devem ser interligados com o sistema central. Essencialmente, você está identificando pontos de junção em um programa onde os aspectos serão compostos.
4. *Análise resolução de conflitos.* Um problema com aspectos é que eles podem interferir uns com os outros quando são compostos com o sistema central. Conflitos ocorrem quando há um choque de ponto de corte com aspectos diferentes especificando que devem ser compostos no mesmo ponto do programa. No entanto, pode haver conflitos mais sutis. Quando os aspectos são projetados independentemente, eles podem fazer suposições sobre a funcionalidade do sistema central que deve ser modificada. No entanto, quando vários aspectos são compostos, um aspecto pode afetar a funcionalidade do sistema de uma forma não prevista por outros aspectos. O comportamento geral do sistema pode não ser conforme o esperado.

Figura 21.7 Casos de uso de extensão

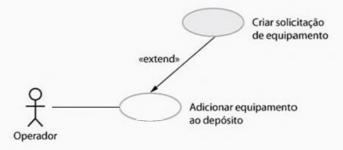
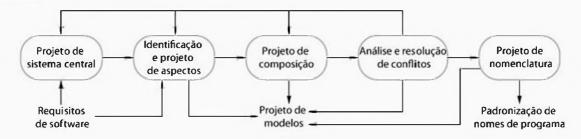



Figura 21.8 Um processo genérico de projeto orientado a aspectos

5. *Projeto de nomenclatura*. Essa é uma atividade importante de projeto que define os padrões para a nomenclatura de entidades no programa. Isso é essencial para evitar problemas de pontos de corte acidentais, os quais ocorrem quando, em algum ponto de junção do programa, o nome acidentalmente combina com o padrão de um ponto de corte. O adendo será aplicado nesse ponto involuntariamente. É óbvio que isso é indesejável e pode gerar um comportamento inesperado do programa. Portanto, você deve projetar um esquema de nomenclatura que minimize a possibilidade de isso acontecer.

Esse processo é, naturalmente, um processo iterativo no qual você cria propostas iniciais de modelagem e depois as refina à medida que analisa e compreende as questões de projeto. Normalmente, você esperaria para refinar as extensões identificadas nos requisitos para um número maior de requisitos.

A saída do processo de projeto orientado a aspectos é um modelo de projeto orientado a aspectos. Isso pode ser expresso em uma versão estendida da UML que inclui construções novas específicas para aspectos, como as propostas por Clarke e Baniassad (2005) e Jacobson e Ng (2004). Os elementos essenciais de 'UML para aspectos' são um meio de modelar aspectos e de especificar pontos de junção onde o adendo de aspecto deve ser interligado com o sistema central.

A Figura 21.9 é um exemplo de um modelo de projeto orientado a aspectos. Usei o estereótipo de UML para aspecto proposto por Jacobson e Ng. A Figura 21.9 mostra o sistema central para um inventário de serviços emergenciais junto com alguns aspectos que podem ser compostos com esse núcleo. Eu mostrei algumas classes do sistema central e alguns aspectos. Trata-se de uma imagem simplificada. Um modelo completo incluiria mais classes e aspectos. Observe como usei notas de UML para fornecer informação adicional sobre as classes que são cruzadas por alguns aspectos.

A Figura 21.10 é um modelo mais detalhado de um aspecto. Certamente, antes de projetar aspectos você deve ter um projeto do sistema central. Como não tenho espaço para mostrar tudo isso aqui, fiz diversas suposições sobre classes e métodos do sistema central.

Figura 21.9 Um modelo de projeto orientado a aspectos

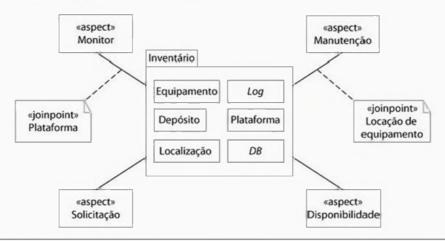
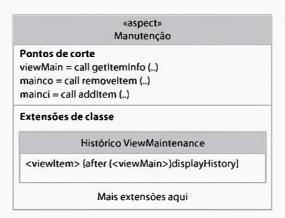



Figura 21.10 Parte do modelo de um aspecto

A primeira seção do aspecto define os pontos de corte que especificam onde ele será composto com o sistema central. Por exemplo, o primeiro ponto de corte especifica que o aspecto pode ser composto na chamada de ponto de junção getItemInfo (..). A seção seguinte define as extensões que são implementadas pelo aspecto. No exemplo, a declaração de extensão pode ser lida como:

No método viewItem, depois de chamar o método getItemInfo, uma chamada para o método displayHistory deve ser incluída para exibir o registro de manutenção.

A programação orientada a aspectos (AOP) teve seu início nos laboratórios de Xerox PARC em 1997, com o desenvolvimento da linguagem de programação AspectJ. Esta ainda é a linguagem orientada a aspectos mais usada, embora extensões orientadas a aspectos de outras linguagens, como C# e C++, também tenham sido implementadas. Outras linguagens experimentais também foram desenvolvidas para suportar separação explícita de interesses e composição de interesses, e existem implementações experimentais de AOP no *framework* .NET. A programação orientada a aspectos é explicada com mais detalhes em outros livros (COLYER et al., 2005; GRADECKI e LEZEIKI, 2003; LADDAD, 2003b).

Se você seguiu uma abordagem orientada a aspectos para projetar seu sistema, já terá identificado a funcionalidade central e as extensões dessa funcionalidade a serem implementadas como aspectos transversais. O foco do processo de programação deve ser, então, escrever o código que implementa funcionalidade central e de extensão e, criticamente, especificar os pontos de corte para que o adendo de aspecto seja composto com o código-base em lugares corretos.

Especificar pontos de corte corretamente é muito importante, uma vez que eles definem onde o adendo de aspecto será composto com a funcionalidade central. Se você cometer um erro na especificação de pontos de corte, o adendo de aspecto será composto com o programa no lugar errado. Isso poderia levar a comportamento inesperado e imprevisível do programa. Aderir aos padrões de nomenclatura estabelecidos durante o projeto de sistema é essencial. Você também deve revisar todos os aspectos para garantir que não ocorra interferência caso dois ou mais aspectos sejam compostos com o sistema no mesmo ponto de junção. Em geral, o melhor é evitar isso completamente, mas, ocasionalmente, pode ser a melhor forma de se implementar um interesse. Nessas circunstâncias, você deve garantir que os aspectos sejam completamente independentes. O comportamento do programa não deve depender da ordem em que os aspectos são compostos no programa.

21.3.3 Verificação e validação

Conforme discutido no Capítulo 8, verificação e validação (V&V) é o processo de se demonstrar que um programa atende a sua especificação (verificação) e às necessidades reais de seus *stakeholders* (validação). Técnicas de verificação estática focam na análise manual ou automatizada do código-fonte do programa. Usa-se validação dinâmica ou teste para descobrir defeitos no programa ou para demonstrar que o programa atende a seus requisitos. Quando a detecção de defeitos é o objetivo, o processo de teste pode ser guiado pelo conhecimento do código-fonte do programa. As métricas de cobertura de teste mostram a eficiência dos testes em fazer com que as declarações do código-fonte sejam executadas.

Para sistemas orientados a aspectos, os processos de testes de validação não são diferentes dos outros sistemas. O programa executável final é tratado como uma caixa preta, e os testes são planejados para mostrar se o sistema atende ou não a seus requisitos. No entanto, o uso de aspectos causa problemas reais com a inspeção de programa e testes de caixa branca, onde o código-fonte do programa é usado para identificar testes potenciais de defeitos.

A inspeção de programas, descrita no Capítulo 24, envolve uma equipe de leitores analisando o código-fonte de um programa para descobrir defeitos que foram introduzidos pelo programador. Trata-se de uma técnica muito eficiente para descobrir defeitos. No entanto, os programas orientados a aspectos não podem ser lidos sequencialmente (ou seja, de cima para baixo). Por isso, tornam-se difíceis de serem entendidos pelas pessoas.

Uma diretriz geral para o entendimento de programas é que o leitor deve ser capaz de ler um programa da esquerda para a direita ou de cima para baixo sem ter de prestar atenção em outras partes do código. Isso torna o entendimento mais fácil para leitores, além de fazer com que os programadores cometam menos erros, uma vez que sua atenção é focada em uma única seção do programa. Melhorar o entendimento dos programas foi o principal motivo para a introdução de programação estruturada (DIJKSTRA et al., 1972) e a eliminação de declarações de desvio incondicional (*go-to*) das linguagens de programação de alto nível.

Em um sistema orientado a aspectos, a leitura sequencial de código é impossível. O leitor precisa examinar cada aspecto, entender seus pontos de corte (os quais podem ser padrões) e o modelo de pontos de junção da

linguagem orientada a aspectos. Ao ler o programa, o leitor deve identificar cada ponto de junção potencial e redirecionar atenção para o código de aspecto para ver se pode ser composto nesse ponto. Sua atenção retorna, então, ao fluxo principal do código-base. Na prática, isso se torna impossível, e a única maneira possível para inspecionar um programa orientado a aspectos é com o uso de ferramentas de leitura de código.

As ferramentas de leitura de código podem ser escritas para 'achatar' um programa orientado a aspectos e apresentar o programa ao leitor com aspectos compostos com o programa em pontos específicos de junção. No entanto, essa não é uma solução completa para o problema da leitura de código. O modelo de pontos de junção em uma linguagem de programação orientada a aspectos pode ser dinâmico em vez de estático, e pode ser impossível demonstrar que o programa achatado se comportará exatamente da mesma forma como o programa que vai executar. Além disso, por ser possível que os aspectos diferentes tenham a mesma especificação do ponto de corte, a ferramenta de leitura de programa deve saber como o compositor de aspectos lida com essa 'competição' de aspectos e como a composição será ordenada.

Teste estrutural ou de caixa branca é uma abordagem sistemática para testes nos quais o conhecimento do código-fonte do programa é usado para projetar testes de defeitos. O objetivo é desenvolver testes que forneçam algum nível de cobertura do programa. Ou seja, o conjunto de testes deve garantir que todo caminho lógico do programa seja executado, com a consequência de que cada declaração do programa seja executada ao menos uma vez. Analisadores de execução de programas podem ser usados para demonstrar que esse nível de cobertura de testes foi alcançado.

Em um sistema orientado a aspectos, existem dois problemas com essa abordagem:

1. Como o conhecimento do código-fonte de programa pode ser usado para derivar sistematicamente os testes do programa?
2. O que de fato significa cobertura de teste?

Para projetar testes em um programa estruturado (por exemplo, testes do código de um método) sem desvios incondicionais, você pode derivar um fluxograma a partir do programa, que revela todos os caminhos de execução lógica através do programa. Você pode, então, examinar o código e, para cada caminho através do fluxograma, escolher os valores de entrada que farão com que o caminho seja executado.

No entanto, um programa orientado a aspectos não é um programa estruturado. O fluxo de controle é interrompido por declarações do tipo 'come from' (CONSTANTINOS et al., 2004). Em algum ponto de junção na execução do código-base, um aspecto pode ser executado. Não tenho certeza se é possível construir um diagrama estruturado de fluxo para tal situação. Dessa forma, fica difícil projetar sistematicamente testes de programa que garantam que todas as combinações de código-base e aspectos sejam executadas.

Em um programa orientado a aspectos, existe também o problema de se decidir o que significa 'cobertura de testes'. Isso significa que o código de cada aspecto será executado ao menos uma vez? Essa é uma condição muito fraca por causa da interação entre aspectos e código-base nos pontos de junção em que os aspectos são compostos. Será que a ideia de cobertura de testes deve ser estendida para que o código do aspecto seja executado ao menos uma vez em cada ponto de junção especificado no ponto de corte do aspecto? O que acontece em tais situações se aspectos diferentes definirem o mesmo ponto de corte? Ambos são problemas teóricos e práticos. Precisamos de ferramentas que suportem testes de programas orientados a aspectos e que ajudarão a avaliar a extensão da cobertura de testes de um sistema.

Como é discutido no Capítulo 24, grandes projetos geralmente têm uma equipe de garantia de qualidade separada que define os padrões de testes e que requer uma garantia formal de que revisões e testes de programa foram feitos de acordo com esses padrões. Os problemas de inspecionar e derivar testes para programas orientados a aspectos são uma barreira significante para a adoção do desenvolvimento de software orientado a aspectos em grandes projetos de software desse tipo.

Além dos problemas com inspeções e testes de caixa branca, Katz (2005) identificou outros problemas nos testes de programas orientados a aspectos:

1. Como os aspectos devem ser definidos para que os testes para eles possam ser derivados?
2. Como os aspectos podem ser testados independentemente do sistema-base com o qual devem ser compostos?
3. Como a interferência do aspecto pode ser testada? Conforme vimos, a interferência do aspecto ocorre quando dois ou mais aspectos usam a mesma especificação de ponto de corte.
4. Como os testes podem ser projetados para que todos os pontos de junção do programa sejam executados e testes de aspectos adequados sejam aplicados?

Basicamente, esses problemas com testes ocorrem porque os aspectos são fortemente interligados com o código-base de um sistema. Por isso, são difíceis de serem testados isoladamente. Como eles podem ser compostos no programa em vários lugares diferentes, você não pode ter certeza de que um aspecto que funciona corretamente em um ponto de junção funcionará corretamente em todos os pontos de junção. Todos esses continuam sendo problemas a serem pesquisados no desenvolvimento de software orientado a aspectos.

PONTOS IMPORTANTES

- O maior benefício de uma abordagem orientada a aspectos para desenvolvimento de software é que ela suporta a separação de interesses. Ao representar interesses transversais como aspectos, os interesses individuais podem ser entendidos, reusados e modificados sem alterar outras partes do programa.
- Entrelaçamento ocorre quando um módulo do sistema inclui um código que implementa diferentes requisitos de sistema. O fenômeno relacionado de espalhamento ocorre quando a implementação de um único interesse está espalhada através de vários componentes em um programa.
- Aspectos incluem um ponto de corte — uma instrução que define onde o aspecto será inserido no programa — e um adendo — o código que implementa o interesse cruzado. Os pontos de junção são os eventos que podem ser referenciados em um ponto de corte.
- Para garantir a separação de interesses, os sistemas podem ser projetados como um sistema central que implementa os interesses primários dos *stakeholders* e um conjunto de extensões que implementam interesses secundários.
- Para identificar os interesses, você pode usar a abordagem orientada a pontos de vista para engenharia de requisitos para elicitar requisitos de *stakeholders* e para identificar interesses transversais de qualidade de serviços e de políticas.
- A transição de requisitos para projeto pode ser feita com a identificação de casos de uso, em que cada caso de uso representa um interesse dos *stakeholders*. O projeto pode ser modelado a partir de uma versão estendida da UML com estereótipos para aspectos.
- Os problemas de inspeção e derivação de testes para programas orientados a aspectos é uma barreira significante para a adoção do desenvolvimento de software orientado a aspectos em grandes projetos de software.

LEITURA COMPLEMENTAR

"Aspect-oriented programming". Essa edição especial da CACM tem uma série de artigos para o público geral. Os textos são um bom ponto de partida para leitura sobre programação orientada a aspectos. (*Comm. ACM*, v. 44, n. 10, out. 2001.) Disponível em: <http://dx.doi.org/10.1145/383845.383846>.

Aspect-oriented Software Development. Um livro de vários autores contendo uma grande coleção de artigos sobre desenvolvimento de software orientado a aspectos, escrito por muitos dos principais pesquisadores da área. (FILMAN, R. E.; ELRAD, T.; CLARKE, S.; AKSIT, M. *Aspect-oriented Software Development*. Addison-Wesley, 2005.)

Aspect-oriented Software Development with Use Cases. Esse é um livro prático para projetistas de software. Os autores discutem como usar casos de uso para gerenciar a separação de interesses, e como usá-los para base de um projeto orientado a aspectos. (JACOBSON, I.; NG, P. *Aspect-oriented Software Development with Use Cases*. Addison-Wesley, 2005.)

EXERCÍCIOS

21.1 Quais são os diferentes tipos de interesse de *stakeholders* que podem surgir em um sistema de grande porte? Como os aspectos podem apoiar a implementação de cada um desses tipos de interesses?

21.2 Resuma o que significa entrelaçamento e espalhamento. Explique, com exemplos, por que entrelaçamento e espalhamento podem causar problemas quando os requisitos de sistema mudam.

21.3 Qual é a diferença entre um ponto de junção e um ponto de corte? Explique como eles facilitam a composição do código dentro de um programa para lidar com interesses transversais.

21.4 Quais suposições sustentam a ideia de que um sistema deve ser organizado como um sistema central que implementa os requisitos essenciais, mais extensões que implementam funcionalidade adicional? Você poderia pensar em sistemas em que esse modelo não seria adequado?

21.5 Quais pontos de vista devem ser considerados quando se desenvolve uma especificação de requisitos para MHC-PMS? Quais poderiam ser os interesses transversais mais importantes?

21.6 Usando a funcionalidade geral para cada ponto de vista mostrado no Quadro 21.3, identifique mais seis casos de uso para o sistema de inventário de equipamentos, além dos mostrados na Figura 21.6. Quando apropriado, mostre como alguns deles poderiam ser organizados como casos de uso de extensão.

21.7 Usando a notação de estereótipo de aspecto ilustrada na Figura 21.10, desenvolva com mais detalhes os aspectos Solicitação e Monitor da Figura 21.9.

21.8 Explique como pode surgir a interferência de aspectos e sugira o que deve ser feito durante o processo de projeto de sistema para reduzir os problemas de interferência de aspectos.

21.9 Explique por que expressar as definições de pontos de corte na forma de padrões aumenta os problemas com testes de programas orientados a aspectos. Para responder isso, pense em como os testes de programa geralmente envolvem a comparação de saídas esperadas com as saídas geradas pelo programa.

21.10 Sugira como você poderia usar aspectos para simplificar a depuração de programas.

REFERÊNCIAS

BIRRER, I.; PASETTI, A.; ROHLIK, O. *The XWeaver Project*: Aspect-oriented Programming for On-Board Applications. 2005. Disponível em: <http://control.ee.ethz.ch/index.cgi?page=publications;action= details;id=2361>.

CLARK, S.; BANIASSAD, E. *Aspect-Oriented Analysis and Design*: The Theme Approach. Harlow, UK: Addison-Wesley, 2005.

COLYER, A.; CLEMENT, A. Aspect-oriented programming with AspectJ. *IBM Systems J.*, v. 44, n. 2, 2005, p. 301-308.

COLYER, A.; CLEMENT, A.; HARLEY, G.; WEBSTER, M. *Eclipse AspectJ*. Upper Saddle River, NJ: Addison-Wesley, 2005.

CONSTANTINOS, C.; SKOTINIOTIS, T.; STOERZER, T. AOP considered harmful. *European Interactive Workshop on Aspects in Software* (EIWAS'04). Berlim, Alemanha, 2004.

DIJKSTRA, E. W.; DAHL, O. J.; HOARE, C. A. R. *Structured Programming*. Londres: Academic Press, 1972.

EASTERBROOK, S.; NUSEIBEH, B. Using ViewPoints for inconsistency management. *BCS/IEE Software Eng. J.*, v. 11, n. 1, 1996, p. 31-43.

FINKELSTEIN, A.; KRAMER, J.; NUSEIBEH, B.; GOEDICKE, M. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *Int. J. of Software Engineering and Knowledge Engineering*, v. 2, n. 1, 1992, p. 31-58.

GRADECKI, J. D.; LEZEIKI, N. *Mastering AspectJ*: Aspect-Oriented Programming in Java. Nova York: John Wiley & Sons, 2003.

JACOBSEN, I.; NG, P.-W. *Aspect-oriented Software Development with Use Cases*. Boston: Addison-Wesley, 2004.

KATZ, S. A Survey of Verification and Static Analysis for Aspects. 2005. Disponível em: <http://www.aosd-europe.net/documents/verificM81.pdf>.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. Getting Started with AspectJ. *Comm. ACM*, v. 44, n. 10, 2001, p. 59-65.

KOTONYA, G.; SOMMERVILLE, I. Requirements engineering with viewpoints. *BCS/IEE Software Eng. J.*, v. 11, n. 1, 1996, p. 5-18.

LADDAD, R. *AspectJ in Action*. Greenwich, Conn.: Manning Publications Co., 2003a.

_______. *AspectJ in Action*: Practical Aspect-Oriented Programming. Greenwich, Conn.: Manning Publications, 2003b.

SOMMERVILLE, I.; SAWYER, P. Viewpoints: principles, problems and a practical approach to requirements engineering. *Annals of Software Engineering*, v. 3, 1997, p. 101-30.

SOMMERVILLE, I.; SAWYER, P.; VILLER, S. Viewpoints for requirements elicitation: a practical approach. *III Int. Conf. on Requirements Engineering*. Colorado: IEEE Computer Society Press, 1998, p. 74-81.

PARTE

1 2 3 **4**

Gerenciamento de software

Algumas vezes, foi sugerido que a principal diferença entre a engenharia de software e outros tipos de programação é que a engenharia de software é um processo gerenciado. Dessa forma, o desenvolvimento de software ocorre dentro de uma organização, a qual está sujeita a uma série de restrições organizacionais, de orçamento e de cronograma. Portanto, o gerenciamento é muito importante para a engenharia de software. Nesta parte do livro, eu apresento uma variedade de tópicos de gerenciamento centrados em questões de gerenciamento técnico, e não em questões mais 'suaves' de gerenciamento, como gerenciamento de pessoas, ou o gerenciamento mais estratégico de sistemas corporativos.

O Capítulo 22 apresenta o gerenciamento do projeto de software e sua primeira seção trata do gerenciamento de riscos. Assim como o planejamento de projetos, o gerenciamento de riscos, em que os gerentes podem identificar o que pode dar errado e planejar o que se pode fazer sobre isso, é uma importante responsabilidade do gerenciamento do projeto. Esse capítulo também inclui seções sobre gerenciamento de pessoas e trabalho de equipe.

O Capítulo 23 abrange a estimativa e o planejamento de projetos. Além de apresentar gráficos de barras como ferramentas de planejamento fundamentais, explico por que o desenvolvimento dirigido a planos continuará a ser uma abordagem de desenvolvimento importante, apesar do sucesso dos métodos ágeis. Também abrange questões que influenciam o preço cobrado por um sistema e as técnicas de estimativas de custo de software. Eu uso a família COCOMO de modelos de custo para descrever a modelagem algorítmica de custos e explicar as vantagens e desvantagens dessa abordagem.

Os capítulos 24 a 26 discutem as questões de gerenciamento de qualidade. O gerenciamento de qualidade está interessado nos processos e nas técnicas para garantir e melhorar a qualidade do software. Esse tópico é apresentado no Capítulo 24, no qual eu ainda discuto a importância das normas de gerenciamento de qualidade e o uso de revisões e das inspeções do processo de garantia de qualidade, bem como o papel da medição de software no gerenciamento de qualidade.

No Capítulo 25, discuto o gerenciamento de configuração. Essa é uma questão importante para todos os grandes sistemas, os quais são desenvolvidos por equipes. No entanto, a necessidade de gerenciamento de configuração nem sempre é óbvia para os alunos, preocupados apenas com o desenvolvimento de software pessoal; portanto, descrevem-se aqui os diferentes aspectos desse tópico, incluindo planejamento de configurações, gerenciamento de versões, construção de sistemas e gerenciamento de mudanças.

Finalmente, o Capítulo 26 abrange a melhoria do processo de software — como os processos podem ser modificados para melhorar os atributos de produto e de processo? Nele, discuto os estágios de um processo de melhoria de processo genérico, ou seja, medição de processo, análise de processo e mudança de processo. Em seguida, discuto a abordagem do SEI baseada em capacidades para a melhoria de processos e explico rapidamente os modelos de maturidade e de capacitação.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 **22** 23 24 25 26

Gerenciamento de projetos

Objetivos

O objetivo deste capítulo é introduzir o gerenciamento dos projetos de software e duas importantes atividades de gereciamento, chamadas gerenciamento de riscos e gerenciamento de pessoas. Com a leitura deste capítulo, você:

- saberá as principais tarefas dos gerentes de projetos de software;
- conhecerá a noção de gerenciamento de riscos e alguns riscos que podem surgir nos projetos de software;
- entenderá os fatores que influenciam a motivação pessoal e o que eles podem significar para os gerentes de projeto de software;
- compreenderá as questões principais que influenciam um trabalho de equipe, como comunicação, organização e composição de equipe.

Conteúdo

O gerenciamento de projetos de software é uma parte essencial da engenharia de software. Os projetos precisam ser gerenciados, pois a engenharia de software profissional está sempre sujeita a orçamentos organizacionais e restrições de cronograma. O trabalho do gerente de projetos é garantir que o projeto de software atenda e supere essas restrições, além de oferecer softwares de alta qualidade. O sucesso do projeto não é garantido por um bom gerenciamento. No entanto, a mau gerenciamento costuma resultar em falha do projeto — o software pode ser entregue com atraso, custar mais do que o inicialmente estimado, ou não se conseguem satisfazer as expectativas dos clientes.

Os critérios de sucesso para o gerenciamento de projetos, certamente, variam de um projeto para outro, mas, para a maioria dos projetos, estas são as metas mais importantes:

1. Fornecer o software ao cliente no prazo estabelecido.
2. Manter os custos gerais dentro do orçamento.
3. Entregar software que atenda às expectativas do cliente.
4. Manter uma equipe de desenvolvimento que trabalhe bem e feliz.

Esses objetivos não são exclusivos para a engenharia de software, mas são os objetivos de todos os projetos de engenharia. No entanto, a engenharia de software é diferente dos outros tipos de engenharia de muitas formas que tornam o gerenciamento de software particularmente desafiador. Algumas dessas diferenças são:

1. O produto é intangível. Um gerente de uma construção de navio ou de um projeto de engenharia civil pode ver o produto que está sendo desenvolvido. Se ocorre um atraso no cronograma, o efeito sobre o produto é visível — partes da estrutura ficam, obviamente, inacabadas. O software é intangível. Ele não pode ser visto ou tocado.

Os gerentes de projeto de software não podem ver o progresso, simplesmente olhando para o artefato que está sendo construído. Em vez disso, eles dependem de outros para produzir provas que eles possam usar para revisar o progresso do trabalho.

2. Os grandes projetos de software são, muitas vezes, 'projetos únicos'. Geralmente, os grandes projetos de software são diferentes dos projetos anteriores em alguns aspectos. Portanto, até mesmo os gerentes que têm grande experiência prévia podem achar difícil antecipar problemas. Além disso, as rápidas mudanças tecnológicas em computadores e comunicações podem tornar obsoleta a experiência de um gerente. As lições aprendidas em projetos anteriores podem não ser transferíveis para novos projetos.
3. Os processos de softwares são variáveis e de organização específica. Os processos de engenharia para alguns tipos de sistemas, tais como pontes e edifícios, são bem compreendidos. No entanto, os processos de software variam significativamente de uma organização para outra. Embora tenham surgido progressos significativos na padronização e melhorias de processos, ainda não somos confiantes em prever quando um processo de software, em particular, conduzirá a problemas de desenvolvimento, especialmente quando o projeto de software é parte de um projeto de engenharia de sistemas mais amplo.

Não se surpreenda se alguns desses projetos de software estiverem atrasados, acima do orçamento e fora do cronograma, devido a esses problemas. Os sistemas de software costumam ser tecnicamente inovadores. Os projetos de engenharia (como novos sistemas de transportes) que são inovadores frequentemente têm problemas de cronograma. Dadas as dificuldades envolvidas, é notável que tantos projetos de software ainda sejam entregues dentro do prazo e do orçamento!

É impossível fazer uma descrição do trabalho-padrão para um gerente de projeto de software. O trabalho varia muito, de acordo com a organização e o produto de software que está sendo desenvolvido. No entanto, a maioria dos gerentes assume a responsabilidade em algum momento para algumas ou todas as atividades apresentadas a seguir:

1. *Planejamento de projeto.* Os gerentes de projetos são responsáveis por planejamento, elaboração de estimativa e de cronograma de desenvolvimento de projeto e atribuição de algumas tarefas para as pessoas. Eles supervisionam o trabalho para garantir que seja realizado conforme os padrões exigidos e acompanham o progresso realizado para verificar se o desenvolvimento está no prazo e dentro do orçamento.
2. *Geração de relatórios.* Geralmente, os gerentes de projeto são responsáveis pela geração de relatórios sobre o andamento de um projeto para os clientes e para os gerentes da empresa que desenvolve o software. Eles precisam ser capazes de se comunicar em vários níveis, desde informações técnicas detalhadas até resumos de gerenciamento. Eles devem escrever documentos concisos e coerentes que abstraem as informações críticas dos relatórios de projeto detalhados. Eles devem ser capazes de apresentar essas informações durante as revisões de progresso.
3. *Gerenciamento de riscos.* Os gerentes de projeto devem avaliar os riscos que podem afetar um projeto, controlar esses riscos e agir quando surgem problemas.
4. *Gerenciamento de pessoas.* Os gerentes de projeto são responsáveis por gerenciar uma equipe de pessoas. Eles devem escolher as pessoas para sua equipe e estabelecer formas de trabalhar que levem a um desempenho eficaz da equipe.
5. *Elaboração de propostas.* O primeiro estágio de um projeto de software pode envolver o processo de escrever uma proposta visando ganhar um contrato para executar um item de trabalho. A proposta descreve os objetivos do projeto e como ele vai ser realizado. Geralmente, ela inclui estimativas de custo e cronograma e justifica por que o projeto deve ser confiado a uma determinada organização ou equipe. A elaboração da proposta é uma tarefa crítica pois a sobrevivência de muitas empresas de software dependem do aceite de propostas suficientes e de ter vários contratos assinados. Não pode haver um conjunto de diretrizes definidas para essa tarefa; a elaboração de propostas é uma habilidade que você adquire através da prática e da experiência.

Neste capítulo, eu discuto o gerenciamento de riscos e o gerenciamento de pessoas. O planejamento de projeto é um tema importante em si mesmo, o qual é discutido no Capítulo 23.

22.1 Gerenciamento de riscos

O gerenciamento de riscos é um dos trabalhos mais importantes para um gerente de projeto. Ele envolve antecipar os riscos que podem afetar o cronograma do projeto ou a qualidade do software que está sendo desenvolvido e tomar medidas para evitar tais riscos (HALL, 1998; OULD, 1999). Você pode pensar em um risco como algo que você preferiria que não acontecesse. Os riscos podem ameaçar o projeto, o software que está sendo desenvolvido ou a organização. Existem três categorias de risco relacionadas:

1. *Riscos de projeto*. Riscos que afetam o cronograma ou os recursos de projeto. Um exemplo de um risco de projeto é a perda de um projetista experiente. Encontrar um projetista substituto com competência e experiência adequados pode demorar muito tempo e, por conseguinte, o projeto de software vai demorar mais tempo para ser concluído.
2. *Riscos de produto*. Riscos que afetam a qualidade ou o desempenho do software que está sendo desenvolvido. Um exemplo de um risco de produto é a falha de um componente comprado para o desempenho esperado, podendo afetar o desempenho geral do sistema de forma mais lenta do que o esperado.
3. *Riscos de negócio*. Os riscos que afetam a organização que desenvolve ou adquire o software. Por exemplo, um concorrente que introduz um novo produto é um risco empresarial. A introdução de um produto competitivo pode significar que as suposições feitas sobre vendas de produtos de software existentes podem ser excessivamente otimistas.

Naturalmente, esses tipos de riscos se sobrepõem. Se um programador experiente deixa um projeto, esse pode ser um risco de projeto porque, mesmo que ele seja imediatamente substituído, o cronograma será afetado. Inevitavelmente, leva tempo para um novo membro do projeto compreender o trabalho que foi feito; dessa forma, eles não podem ser produtivos imediatamente. Consequentemente, a entrega do sistema pode ser adiada. A perda de um membro da equipe também pode ser um risco de produto porque um substituto pode não ser tão experiente e, assim, cometer erros de programação. Finalmente, pode significar um risco de negócio porque a experiência do programador pode ser crucial para ganhar novos contratos.

Você deve registrar os resultados da análise de risco no plano de projeto, juntamente com uma análise de consequências, a qual estabelece as consequências dos riscos para o projeto, produto e negócio. Um gerenciamento eficaz de riscos torna mais fácil lidar com os problemas para garantir que eles não demandem um orçamento ou um atraso de cronograma inaceitáveis.

Os riscos específicos que podem afetar um projeto dependem do projeto e do ambiente organizacional no qual o software está sendo desenvolvido. No entanto, também há riscos comuns que não estão relacionados com o tipo de software desenvolvido, e estes podem ocorrer em qualquer projeto. A Tabela 22.1 mostra esses riscos.

Um gerenciamento de riscos é particularmente importante para projetos de software por causa das incertezas inerentes que a maioria dos projetos enfrenta. Elas se originam de requisitos vagamente definidos, mudanças de requisitos devido a mudanças nas necessidades de cliente, dificuldades em estimar o tempo e os recursos necessários para o desenvolvimento do software e diferenças nas habilidades individuais. É preciso prever os riscos, compreender o impacto desses riscos sobre o projeto, o produto e o negócio e tomar medidas para evitar tais riscos. Talvez você precise elaborar planos de contingência para que, quando ocorrerem os riscos, você possa tomar medidas de recuperação imediata.

A Figura 22.1 ilustra um esboço do processo de gerenciamento de riscos envolvendo vários estágios:

4. *Identificação de riscos*. Você deve identificar possíveis riscos de projeto, de produto e de negócios.
5. *Análise de riscos*. Você deve avaliar a probabilidade e as consequências desses riscos.
6. *Planejamento de riscos*. Você deve planejar para enfrentar o risco, evitando ou minimizando seus efeitos sobre o projeto.
7. *Monitoramento de riscos*. Você deve avaliar regularmente os riscos e seus planos para mitigação de riscos e atualizá-los quando souber mais sobre os riscos.

É necessário documentar os resultados do processo de gerenciamento de riscos em um plano de gerenciamento de riscos, incluir uma discussão sobre os riscos enfrentados pelo projeto e uma análise desses riscos e obter informações sobre a maneira de gerenciar o risco caso seja provável que eles se tornem um problema.

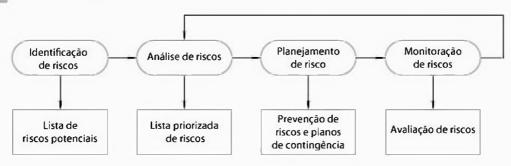
O processo do gerenciamento de riscos é um processo iterativo que continua ao longo do projeto. Depois de elaborar um plano de gerenciamento de riscos inicial, você monitora a situação para detectar riscos emergentes. Obtendo mais informações sobre os riscos e caso estes se tornem disponíveis, você deve reavaliar os riscos e decidir se a prioridade de risco mudou. Você pode precisar mudar seus planos para prevenção de riscos e gerenciamento de contingência.

22.1.1 Identificação de riscos

A identificação de riscos é o primeiro estágio do processo de gerenciamento de riscos. Ele diz a respeito à identificação de riscos que poderiam representar uma grande ameaça para o processo de engenharia de software, o software que está sendo desenvolvido ou a organização de desenvolvimento. A identificação de riscos

Tabela 22.1 Exemplos de riscos comuns de projeto, produto e negócio

Risco	Afeta	Descrição
Rotatividade de pessoal	Projeto	Pessoal experiente deixará o projeto antes de ser concluído.
Mudança de gerência	Projeto	Haverá uma mudança na gerência da organização com prioridades diferentes.
Indisponibilidade de hardware	Projeto	Hardware que é essencial para o projeto não será entregue no prazo.
Mudança de requisitos	Projeto e produto	Haverá um número maior de alterações nos requisitos do que o previsto.
Atrasos de especificação	Projeto e produto	Especificações de interfaces essenciais não estão disponíveis no prazo.
Tamanho subestimado	Projeto e produto	O tamanho do sistema foi subestimado.
Baixo desempenho de ferramenta CASE	Produto	Ferramentas CASE, que apoiam o projeto, não executam como o previsto.
Mudança de tecnologia	Negócio	A tecnologia sobre a qual o sistema foi construído é substituída por uma nova tecnologia.
Concorrência de produto	Negócio	Um produto concorrente é comercializado antes que o sistema seja concluído.

Figura 22.1 Processo de gerenciamento de riscos

pode ser um processo de equipe quando uma equipe se junta para dicutir os riscos possíveis. Como alternativa, o gerente de projeto pode usar sua experiência para identificar os riscos mais prováveis ou críticos.

Um *checklist* de verificação de tipos diferentes de risco pode ser usado como ponto de partida para a identificação de riscos. Existem pelo menos seis tipos de riscos que podem ser incluídos em um *checklist* de verificação de riscos:

1. *Riscos de tecnologia*. Riscos que derivam das tecnologias de software ou hardware que são usadas para desenvolver o sistema.
2. *Riscos de pessoas*. Riscos que são associados às pessoas da equipe de desenvolvimento.
3. *Riscos organizacionais*. Riscos que derivam do ambiente organizacional onde o software está sendo desenvolvido.
4. *Riscos de ferramentas*. Riscos que derivam das ferramentas de software e outros softwares de suporte usados para desenvolver o sistema.
5. *Riscos de requisitos*. Riscos que derivam das mudanças nos requisitos de cliente e no processo de gerenciamento de mudanças de requisitos.
6. *Riscos de estimativas*. Riscos que derivam das estimativas de gerenciamento dos recursos necessários para construir o sistema.

A Tabela 22.2 dá alguns exemplos de possíveis riscos em cada uma dessas categorias. Ao terminar o processo de identificação de riscos, você deve ter uma longa lista de riscos que poderiam ocorrer e afetar o produto, o processo e os negócios. Assim, você precisa reduzir essa lista para um tamanho gerenciável. Se você tiver muitos riscos, é praticamente impossível manter o controle de todos eles.

22.1.2 Análise de riscos

Durante o processo de análise de riscos, você deve considerar cada risco identificado e fazer um julgamento sobre a probabilidade e a gravidade desses riscos. Não há uma maneira fácil de fazer isso. Você tem de confiar em seu próprio julgamento e na experiência advinda de projetos anteriores e de problemas que surgiram neles. Não é possível fazer uma avaliação precisa e numérica da probabilidade e gravidade de cada risco. Em vez disso, você deve atribuir o risco a um entre vários tipos:

1. A probabilidade do risco pode ser avaliada como muito baixa (< 10%), baixa (10 a 25%), moderada (25 a 50%), alta (50 a 75%) ou muito alta (> 75%).
2. Os efeitos do risco podem ser avaliados como catastróficos (ameaçam a sobrevivência do projeto), graves (causariam grandes atrasos), toleráveis (os atrasos estão dentro da contingência permitida) ou insignificantes.

Então, você deve mensurar os resultados desse processo de análise usando uma tabela ordenada de acordo com a gravidade do risco. A Tabela 22.3 ilustra os riscos identificados na Tabela 22.2. Nesse momento ocorre a avaliação arbitrária da probabilidade e da gravidade. Para fazer essa avaliação, você precisa obter informações detalhadas sobre o projeto, o processo, a equipe de desenvolvimento e a organização.

Naturalmente, a probabilidade e a avaliação dos efeitos dos riscos podem mudar à medida que mais informações sobre o risco se tornam disponíveis e quando planos de gerenciamento de riscos são implementados. Portanto, você deve atualizar essa tabela durante cada iteração do processo de risco.

Uma vez que os riscos foram analisados e classificados, você deve avaliar quais desses riscos são mais significativos. Seu julgamento deve depender de uma combinação da probabilidade de ocorrência de risco e dos efeitos desse risco. Em geral, os riscos catastróficos devem ser sempre considerados, bem como os riscos graves que têm mais do que uma probabilidade moderada de ocorrência.

Boehm (1988) recomenda a identificação e a monitoração dos dez principais riscos, mas acredito que esse número é um pouco arbitrário. O número certo de riscos a serem monitorados deve depender do projeto. Podem ser cinco, ou talvez sejam 15. No entanto, o número de riscos escolhidos para monitoração precisa ser gerenciável. Um número muito grande de riscos simplesmente exigiria muita informação para se coletar. Dos riscos identificados na Tabela 22.3, é oportuno considerar os oito riscos que têm consequências catastróficas ou graves (Tabela 22.4).

Tabela 22.2 Exemplos de diferentes tipos de riscos

Tipo de risco	Possíveis riscos
Tecnologia	O banco de dados usado no sistema não pode processar tantas transações por segundo, quanto o esperado. (1) Os componentes de software reusáveis contêm defeitos que significam que eles não podem ser reusados como o planejado. (2)
Pessoas	É impossível recrutar pessoas com as habilidades necessárias. (3) As principais pessoas estão doentes e, nos momentos críticos, não estão disponíveis. (4) O treinamento para o pessoal não está disponível. (5)
Organizacional	A organização é reestruturada para que diferentes gerências sejam responsáveis pelo projeto. (6) Problemas financeiros organizacionais forçam reduções no orçamento de projeto. (7)
Ferramentas	O código gerado pelas ferramentas de geração de código de software é ineficiente. (8) As ferramentas de software não podem trabalhar juntas de forma integrada. (9)
Requisitos	São propostas mudanças nos requisitos que requerem retrabalho de projeto. (10) Os clientes não conseguem compreender o impacto das mudanças nos requisitos. (11)
Estimativa	O tempo necessário para desenvolver o software é subestimado. (12) A taxa de reparo de defeito é subestimada. (13) O tamanho do software é subestimado. (14)

Tabela 22.3 Exemplos e tipos de riscos

Risco	Probabilidade	Efeito
Problemas financeiros organizacionais forçam reduções no orçamento de projeto. (7)	Baixa	Catastrófico
É impossível recrutar pessoal com as habilidades necessárias para o projeto. (3)	Alta	Catastrófico
Pessoas-chave estão doentes nos momentos críticos do projeto. (4)	Moderada	Grave
Defeitos em componentes reusáveis de software precisam ser reparados antes que esses componentes sejam reusados. (2)	Moderada	Grave
Mudanças de requisitos que exigem muito retrabalho de projeto são propostas. (10)	Moderada	Grave
A organização é reestruturada para que gerências diferentes sejam responsáveis pelo projeto. (6)	Alta	Grave
O banco de dados usado no sistema não pode processar tantas transações por segundo quanto o esperado. (1)	Moderada	Grave
O tempo necessário para desenvolver o software é subestimado. (12)	Alta	Grave
Ferramentas de software não podem ser integradas. (9)	Alta	Tolerável
Os clientes não conseguem compreender o impacto das mudanças de requisitos. (11)	Moderada	Tolerável
Treinamento necessário de pessoal não está disponível. (5)	Moderada	Tolerável
A taxa de reparo de defeitos é subestimada. (13)	Moderada	Tolerável
O tamanho do software é subestimado. (14)	Alta	Tolerável
O código gerado por ferramentas de geração de código é ineficiente. (8)	Moderada	Insignificante

Tabela 22.4 Estratégias para ajudar a gerenciar os riscos

Risco	Estratégia
Problemas financeiros organizacionais	Prepare um documento de informações essenciais para a gerência sênior mostrando como o projeto está fazendo uma contribuição muito importante para os objetivos do negócio e apresentando os motivos pelos quais os cortes no orçamento não seriam efetivos.
Problemas de recrutamento	O cliente alerta para possíveis dificuldades e a possibilidade de atrasos; investigue as compras de componentes.
Doença de pessoal	Reorganize a equipe para que ocorra mais sobreposição de trabalho e, portanto, as pessoas compreendam o trabalho umas das outras.
Componentes defeituosos	Substitua componentes potencialmente defeituosos comprando componentes de confiabilidade conhecida.
Mudanças de requisitos	Derive informações de rastreabilidade para avaliar o impacto das mudanças de requisitos; maximize o ocultamento de informações no projeto.
Reestruturação organizacional	Prepare um documento de informações essenciais para a gerência sênior mostrando como o projeto está fazendo uma contribuição muito importante para os objetivos do negócio.
Desempenho de banco de dados	Investigue a possibilidade de comprar um banco de dados de alto desempenho.
Prazo de desenvolvimento subestimado	Investigue as compras de componentes; investigue o uso de um gerador de programas.

22.1.3 O planejamento de riscos

O processo de planejamento de riscos considera cada um dos principais riscos que foram identificados e desenvolve estratégias para gerenciar esses riscos. Para cada um dos riscos, você precisa pensar em ações que possa tomar para minimizar a interrupção do projeto, caso ocorra o problema identificado no risco. Você também precisa pensar sobre as informações que pode precisar coletar durante a monitoração do projeto de maneira que os problemas possam ser antecipados. Não existe um processo simples que possa ser seguido para o planejamento de contingência. Ele usa o julgamento e a experiência do gerente de projeto.

A Tabela 22.4 mostra as possíveis estratégias de gerenciamento de riscos identificados para os principais riscos (ou seja, aqueles que são graves ou intoleráveis) mostrados na Tabela 22.3. Essas estratégias estão divididas em três categorias:

1. *Estratégias de prevenção*. Seguir essas estratégias indica que a probabilidade de um risco ocorrer será reduzida. Um exemplo de uma estratégia de prevenção de riscos é a estratégia para lidar com componentes defeituosos, mostrada na Tabela 22.4.
2. *Estratégias de minimização*. Seguir essas estratégias indica que o impacto do risco será reduzido. Um exemplo de uma estratégia de minimização de risco é a estratégia para a doença de pessoal mostrada na Tabela 22.4.
3. *Planos de contingência*. Seguir essas estratégias indica que você está preparado e tem uma estratégia para lidar com o pior. Um exemplo de uma estratégia de contingência é a estratégia para problemas financeiros organizacionais que são mostrados na Tabela 22.4.

Aqui, você pode ver uma analogia clara com as estratégias usadas em sistemas críticos para garantir a confiabilidade, proteção e segurança, em que você deve evitar, tolerar ou se recuperar das falhas. Obviamente, é melhor usar uma estratégia que evite o risco. Se isso não for possível, você deve usar uma estratégia que reduza as chances de riscos com efeitos graves. Finalmente, você deve ter estratégias preparadas para lidar com os riscos caso eles apareçam. Elas devem reduzir o impacto global de um risco relativo ao projeto ou produto.

22.1.4 Monitoração de riscos

A monitoração de riscos é o processo de verificar se suas suposições sobre os riscos de produto, de processo e de negócios não mudaram. Você deve avaliar regularmente cada um dos riscos identificados para decidir se esse risco está se tornando mais ou menos provável. Você também precisa verificar se os efeitos dos riscos mudaram ou não. Para fazer isso, você deve observar outros fatores, como o número de solicitações de mudanças de requisitos, que lhe dão pistas sobre a probabilidade de risco e seus efeitos. Esses fatores dependem, obviamente, dos tipos de riscos. A Tabela 22.5 apresenta alguns exemplos dos fatores que podem ser úteis na avaliação desses tipos de risco.

Você deve monitorar esses riscos, regularmente, em todas as fases de um projeto. Em cada revisão da gerência, você deve considerar e discutir cada um dos principais riscos separadamente. Você deve decidir se os riscos são mais ou menos suscetíveis de surgirem e se a seriedade e as consequências deles se alteraram.

Tabela 22.5 Os indicadores de riscos

Tipo de risco	Potenciais indicadores
Tecnologia	O atraso na entrega de hardware ou software de suporte; muitos relataram problemas de tecnologia.
Pessoas	Pessoas com pouca motivação; relacionamentos fracos entre os membros da equipe; alta rotatividade de pessoal.
Organizacional	Boatos organizacionais; falta de ação da gerência sênior.
Ferramentas	Relutância dos membros da equipe em usar as ferramentas; reclamações sobre as ferramentas CASE; demanda por estações de trabalho mais poderosas.
Requisitos	Muitas solicitações de mudanças de requisitos; reclamações dos clientes.
Estimativa	Falha em cumprir o cronograma aprovado; falha em eliminar os defeitos relatados.

22.2 Gerenciamento de pessoas

As pessoas que trabalham em uma organização de software são seus maiores ativos. Custa muito para recrutar e reter boas pessoas e cabe aos gerentes de software garantirem que a empresa obtenha o melhor retorno possível sobre seus investimentos. Nas empresas e economias de sucesso, isso é alcançado quando as pessoas são respeitadas pelas organizações e lhes são atribuídas responsabilidades que refletem suas habilidades e experiência.

É importante que os gerentes de projeto de software compreendam as questões técnicas que influenciam o trabalho de desenvolvimento de software. Infelizmente, no entanto, os bons engenheiros de software não necessariamente são bons gerentes de pessoas. Frequentemente, os bons engenheiros de software possuem fortes habilidades técnicas, mas podem não ter as habilidades mais flexíveis que lhes permitam motivar e liderar uma equipe de desenvolvimento de projeto. Como gerente de projeto, você deve estar ciente dos potenciais problemas de gerenciamento de pessoas e deve tentar desenvolver competências de gerenciamento de pessoas.

Em minha opinião, há quatro fatores críticos no gerenciamento de pessoas:

1. *Consistência.* Pessoas em uma equipe de projeto devem ser tratadas da mesma forma. Ninguém espera que o reconhecimento seja igual para todos, mas as pessoas não devem sentir que sua contribuição para a organização é subvalorizada.
2. *Respeito.* Pessoas diferentes têm habilidades diferentes e os gerentes devem respeitar essas diferenças. A todos os membros da equipe devem ser dadas oportunidades de contribuir. Claro que, em alguns casos, você vai encontrar pessoas que simplesmente não se encaixam em uma equipe e não podem continuar, mas é importante não tirar conclusões no estágio inicial do projeto.
3. *Inclusão.* Pessoas contribuem efetivamente quando sentem que são ouvidas por outras pessoas e que suas propostas são levadas em consideração. É importante desenvolver um ambiente de trabalho em que todas as visões são consideradas, mesmo aquelas dos membros mais novos da equipe.
4. *Honestidade.* Como gerente, você deve sempre ser honesto sobre o que está indo bem e o que vai mal na equipe. Você também deve ser honesto sobre seu nível de conhecimento técnico e estar disposto a submeter-se a algum membro da equipe com mais conhecimento, quando necessário. Se tentar encobrir sua ignorância ou problemas, você será eventualmente descoberto e perderá o respeito do grupo.

Eu considero que o gerenciamento de pessoas é algo que deve ser baseado na experiência, em vez de aprendido em um livro. Meu objetivo nesta seção e na próxima, sobre trabalho de equipe, é introduzir alguns dos maiores problemas envolvendo o gerenciamento de pessoas e equipes, que afetam o gerenciamento de projetos de software. Espero que o material aqui o sensibilize para alguns dos problemas que os gerentes podem encontrar ao lidar com equipes de indivíduos tecnicamente talentosos.

22.2.1 Motivação de pessoas

Como gerente de projetos, você precisa motivar as pessoas que trabalham com você para que elas contribuam com o melhor de suas habilidades. Motivação significa organizar o trabalho e o ambiente de trabalho para incentivar as pessoas a trabalharem do modo mais eficaz possível. Se as pessoas não são motivadas, elas não se interessarão pelo trabalho que estão fazendo. Elas trabalharão lentamente, serão mais propensas a cometer erros e não contribuirão para os objetivos mais amplos da equipe ou organização.

Para fornecer esse incentivo, você deve entender um pouco sobre o que motiva as pessoas. Maslow (1954) sugere que as pessoas são motivadas por satisfazer suas necessidades. Essas necessidades são organizadas em uma série de níveis, conforme mostrado na Figura 22.2. Os níveis inferiores dessa hierarquia representam as necessidades básicas de alimentação, sono e assim por diante, bem como a necessidade de se sentir seguro em um ambiente. As necessidades sociais abrangem a necessidade de se sentir fazendo parte de um agrupamento social. A necessidade de autoestima representa a necessidade de se sentir respeitado pelos outros, e necessidades de autorrealização estão preocupadas com o desenvolvimento pessoal. As pessoas precisam satisfazer suas necessidades básicas como a fome antes das necessidades mais abstratas e de nível mais alto.

Geralmente, as pessoas que trabalham em organizações de desenvolvimento de software não estão famintas ou sedenas ou fisicamente ameaçadas pelo ambiente. Portanto, é importante, do ponto de vista de gerenciamento, certificar-se de que as necessidades sociais, de autoestima e autorrealização das pessoas são satisfeitas.

Figura 22.2 Hierarquia de necessidades humanas

1. Para satisfazer as necessidades sociais, você precisa dar às pessoas tempo para encontrarem seus colegas de trabalho e oferecer-lhes oportunidades para se encontrarem. Isso é relativamente fácil quando todos os membros de uma equipe de desenvolvimento trabalham no mesmo lugar, mas, cada vez mais, os membros de equipe não estão localizados no mesmo edifício ou na mesma cidade ou Estado. Eles podem trabalhar para diferentes organizações ou, na maioria das vezes, em casa.

   Os sistemas de rede social e teleconferência podem ser usados para facilitar a comunicação, mas a minha experiência com sistemas eletrônicos revela que eles são mais eficazes se as pessoas se conhecerem. Portanto, é importante você organizar alguns encontros presenciais no início do projeto para que as pessoas possam interagir diretamente com outros membros da equipe. Por meio dessa interação direta, as pessoas tornam-se parte de um grupo social e aceitam as metas e prioridades desse grupo.
2. Para satisfazer as necessidades de autoestima, você precisa mostrar às pessoas que elas são valorizadas pela organização, e uma forma simples e eficaz de fazer isso é pelo reconhecimento público. Certamente, as pessoas também devem sentir que são pagas em um nível que reflete suas habilidades e experiência.
3. Finalmente, para satisfazer as necessidades de autorrealização, você precisa dar às pessoas responsabilidade por seu trabalho, atribuir-lhes tarefas exigentes (mas não impossíveis) e fornecer um programa de treinamento em que elas possam desenvolver suas habilidades. O treinamento é uma importante influência motivadora para as pessoas, assim como a aquisição de novos conhecimentos e novas habilidades.

O Quadro 22.1 ilustra um problema de motivação que os gerentes muitas vezes têm de enfrentar. Nesse exemplo, um competente membro do grupo perde o interesse pelo trabalho e pelo grupo como um todo. A qualidade de seu trabalho cai e torna-se inaceitável. Essa situação precisa ser tratada rapidamente. Se você não resolver o problema, outros membros do grupo se tornarão insatisfeitos e sentirão que estão fazendo uma parte injusta do trabalho.

Quadro 22.1 Motivações individuais

Estudo de caso: Motivação

Alice é uma gerente de projetos de software que trabalha em uma empresa que desenvolve sistemas de alarme. Essa empresa pretende entrar no mercado crescente de tecnologia assistencial para ajudar pessoas idosas e com deficiências a viver de forma independente. Alice foi solicitada para liderar uma equipe de seis desenvolvedores para desenvolver novos produtos baseados na tecnologia de alarme da empresa.

O projeto de tecnologia assistencial de Alice começa bem. Bons relacionamentos de trabalho se estabelecem dentro da equipe e surgem novas e criativas ideias. A equipe decide desenvolver um sistema de mensagens ponto-a-ponto usando televisores digitais ligados à rede de alarme para comunicações. No entanto, faltando alguns meses para o projeto, Alice nota que Dorothy, uma especialista em projeto de hardware, começa a chegar atrasada no trabalho, a qualidade de seu trabalho se deteriora cada vez mais e ela não parece estar se comunicando com os outros membros da equipe.

Alice fala sobre o problema informalmente com outros membros da equipe para tentar descobrir se houve mudanças em circunstâncias pessoais de Dorothy e se isso pode afetar seu trabalho. Eles não sabem de nada, então Alice decide falar com Dorothy para tentar entender o problema.

Depois de inicialmente desmentir que existe um problema, Dorothy admite que ela perdeu o interesse no trabalho. Ela esperava que fosse capaz de desenvolver e usar suas habilidades de interfaceamento de hardware, no entanto, por causa da direção escolhida para o produto, ela tem poucas oportunidades para isso. Basicamente, ela está trabalhando como uma programadora C com outros membros da equipe.

Embora admita que o trabalho seja desafiador, ela está preocupada em não desenvolver suas habilidades de interface. Ela está preocupada em encontrar um trabalho que envolva a interface de hardware após esse projeto. Ela decidiu minimizar a conversa com os membros da equipe por não querer perturbá-los com o que está pensando sobre o próximo projeto.

Nesse exemplo, Alice tenta descobrir quais circunstâncias pessoais de Dorothy poderiam ser o problema. Geralmente, as dificuldades pessoais afetam a motivação, pois as pessoas não conseguem se concentrar em seu trabalho. Você pode ter de dar-lhes tempo e suporte para resolver esses problemas, embora você também precise deixar bem claro que eles ainda têm responsabilidades para com seu empregador.

O problema da motivação de Dorothy é bastante comum quando o desenvolvimento do projeto toma uma direção inesperada. Pessoas que esperam fazer certo tipo de trabalho podem acabar fazendo algo completamente diferente. Isso se torna um problema quando os membros da equipe desejam desenvolver suas competências de forma diferente da tomada pelo projeto. Nessas circunstâncias, você pode decidir que o membro da equipe deve deixar a equipe e encontrar oportunidades em outros lugares. Nesse exemplo, no entanto, Alice decide tentar convencer Dorothy que expandir sua experiência é um passo positivo em sua carreira. Ela dá mais autonomia de projeto para Dorothy e organiza cursos de formação em engenharia de software que lhe dará mais oportunidades após terminar o projeto atual.

O modelo de motivação de Maslow é útil até certo ponto, mas acredito que seu problema é ter um ponto de vista exclusivamente pessoal sobre a motivação. Ele não considera, adequadamente, o fato de que as pessoas se sentem parte de uma organização, de um grupo de profissionais ou de uma ou mais culturas. Essa não é apenas uma questão sobre a satisfação de necessidades sociais — as pessoas podem ser motivadas por ajudarem um grupo a atingir seus objetivos comuns.

Tornar-se membro de um grupo coeso é altamente motivador para a maioria das pessoas. As pessoas com empregos satisfatórios, muitas vezes, gostam de ir trabalhar porque são motivadas pelas pessoas com quem trabalham e pelo trabalho que realizam. Portanto, assim como pensar sobre a motivação individual, você também precisa pensar em como um grupo, como um todo, pode ser motivado a atingir os objetivos da organização. Na próxima seção eu discuto as questões de gerenciamento de grupos.

Os tipos de personalidade também influenciam na motivação. Bass e Dunteman (1963) classificam os profissionais em três tipos:

1. *Pessoas orientadas a tarefas*, que são motivadas pelo trabalho que fazem. Na engenharia de software, essas são as pessoas motivadas pelo desafio intelectual do desenvolvimento de software.
2. *Pessoas automotivadas*, que são motivadas, principalmente, pelo sucesso e reconhecimento pessoal. Elas estão interessadas no desenvolvimento de software como um meio de alcançarem seus próprios objetivos. Isso não significa que sejam egoístas e pensem apenas em seus próprios interesses. Em vez disso, frequentemente, essas pessoas têm objetivos de longo prazo que as motivam, como progressão na carreira, e desejam ser bem-sucedidas em seu trabalho para concretizar tais objetivos.
3. *Pessoas orientadas a interações*, que são motivadas pela presença e ações dos colegas de trabalho. Como o desenvolvimento de software é mais centrado no usuário, os indivíduos orientados a interações estão se tornando mais envolvidos na engenharia de software.

Geralmente, as personalidades orientadas a interações gostam de trabalhar como parte de um grupo, ao mesmo tempo em que as pessoas orientadas a tarefas e automotivadas em geral preferem agir individualmente. As mulheres são mais suscetíveis de serem orientadas a interações do que os homens. Geralmente, elas são comunicadoras mais eficazes. No Quadro 22.3, eu discuto no estudo de caso a mistura desses diferentes tipos de personalidades em grupos.

A motivação de cada indivíduo é constituída por elementos de cada classe, mas normalmente um tipo de motivação é dominante em um determinado momento. No entanto, os indivíduos podem mudar. Por exemplo, pessoas técnicas que sentem que não estão sendo devidamente recompensadas podem tornar-se automotivadas e colocar seus interesses pessoais antes de interesses técnicos. Se um grupo funciona particularmente bem, as pessoas auto-orientadas podem tornar-se mais orientadas a interações.

22.3 Trabalho de equipe

A maioria dos softwares profissionais é desenvolvida por equipes de projeto que variam em tamanho, desde duas até várias centenas de pessoas. Como é impossível para todos os membros de um grupo grande trabalharem juntos em um único problema, as grandes equipes são geralmente divididas em vários grupos. Cada grupo é responsável pelo desenvolvimento de parte do sistema global. Como regra geral, os grupos de projetos de engenharia de software não devem ter mais de dez membros. Quando são usados pequenos grupos, os problemas

de comunicação são reduzidos. Todos se conhecem e todo o grupo pode se sentar em torno de uma mesa de reuniões para discutir o projeto e o software que estão desenvolvendo.

Montar uma equipe que tenha o equilíbrio entre as habilidades técnicas, a experiência e as personalidades é uma tarefa de gerenciamento crítico. No entanto, os grupos bem-sucedidos são mais do que um conjunto de indivíduos com equilíbrio de habilidades. Um bom grupo é coeso e tem espírito de equipe. As pessoas envolvidas são motivadas pelo sucesso do grupo, bem como por seus próprios objetivos pessoais.

Os membros de um grupo coeso pensam que o grupo é mais importante do que seus indivíduos. Os membros de um grupo coeso e bem liderado são leais ao grupo. Eles se identificam com os objetivos do grupo e com os outros membros do grupo. Eles tentam proteger o grupo das interferências externas como uma entidade. Isso torna o grupo forte e capaz de lidar com problemas e situações inesperadas.

Os benefícios da criação de um grupo coeso são:

1. *O grupo pode estabelecer seus próprios padrões de qualidade.* Como esses padrões são estabelecidos por consenso, eles são mais propensos a serem observados do que os padrões externos impostos ao grupo.
2. *As pessoas se apoiam e aprendem umas com as outras.* Em um grupo, as pessoas aprendem umas com as outras. Inibições causadas pela ignorância são minimizadas uma vez que a aprendizagem mútua é encorajada.
3. *O conhecimento é compartilhado.* Se um membro deixa o grupo, outros membros devem poder assumir tarefas críticas e garantir que o projeto não seja indevidamente interrompido.
4. *Refatoração e melhorias contínuas são incentivadas.* Os membros do grupo trabalham coletivamente para entregar resultados de alta qualidade e corrigir problemas, independentemente dos indivíduos que originalmente criaram o projeto ou o programa.

Os bons gerentes de projeto sempre devem tentar incentivar a coesão do grupo. Eles podem organizar eventos sociais para os membros do grupo e suas famílias, além de tentar estabelecer um senso de identidade para o grupo, nomeando e estabelecendo uma identidade e território para o grupo, ou eles podem envolver-se em atividades explícitas de construção de grupos, como esportes e jogos.

Uma das maneiras mais eficazes de promoção da coesão é ser inclusivo. Isso significa que você deve tratar os membros do grupo como responsáveis e confiáveis e disponibilizar informações livremente. Às vezes, os gerentes sentem que não podem revelar certas informações para todos no grupo. Invariavelmente, isso cria um clima de desconfiança. A troca de informações é uma maneira eficaz de fazer as pessoas se sentirem valorizadas e como membros de um grupo.

No estudo de caso do Quadro 22.2 você tem um exemplo dessa situação. Regularmente, Alice organiza reuniões informais nas quais ela diz aos outros membros do grupo o que está acontecendo. Ela faz questão de envolver as pessoas no desenvolvimento do produto, pedindo que tragam ideias novas provenientes de suas próprias experiências familiares. Os 'dias fora' também são boas maneiras de promover a coesão — as pessoas relaxam juntas enquanto ajudam umas às outras a aprenderem sobre novas tecnologias.

A eficácia de um grupo depende, em certa medida, da natureza do projeto e da organização que realiza o trabalho. Se uma organização estiver em um estado de agitação com reorganizações constantes e ambiente de insegurança, torna-se muito difícil para os membros da equipe se concentrarem no desenvolvimento de software. No entanto, apesar de questões de projeto e organizacionais, existem três fatores genéricos que afetam o trabalho de equipe:

Quadro 22.2 Coesão do grupo

Estudo de caso: Espírito de equipe

Alice, uma experiente gerente de projetos, compreende a importância da criação de um grupo coeso. Quando eles desenvolvem um novo produto, ela tem a oportunidade de envolver todos os membros do grupo na especificação e no projeto de produto, levando-os a discutirem as possíveis tecnologias com os membros mais idosos de suas famílias. Ela os incentiva a trazerem esses membros da família para encontarem outros membros do grupo de desenvolvimento.

Alice também organiza almoços mensais para todos no grupo. Esses almoços são uma oportunidade para todos os membros da equipe se encontrarem informalmente, conversarem sobre suas questões de interesse e conhecerem uns aos outros. No almoço, Alice diz ao grupo o que ela sabe sobre as notícias organizacionais, políticas, estratégias e assim por diante. Em seguida, cada membro da equipe resume brevemente o que eles têm feito e o grupo discute um tema geral, como as ideias dos parentes idosos para novos produtos.

De tempos em tempos, Alice organiza um 'tempo fora' para o grupo, no qual a equipe passa dois dias 'atualizando as tecnologias'. Cada membro da equipe prepara uma atualização sobre uma tecnologia relevante e apresenta ao grupo. Essa é uma reunião fora do local de trabalho, em um bom hotel, com todo o tempo programado para discussões e interação social.

1. *As pessoas no grupo*. Você precisa de uma mistura de pessoas em um grupo de um projeto, pois o desenvolvimento de software envolve diversas atividades, como negociação com clientes, programação, testes e documentação.
2. *A organização de grupo*. Um grupo pode ser organizado de maneira que os indivíduos possam contribuir com o melhor de suas habilidades e as tarefas possam ser concluídas conforme o esperado.
3. *Comunicações técnicas e gerenciais*. A boa comunicação entre os membros do grupo, bem como entre a equipe de engenharia de software e outros participantes do projeto, é essencial.

Assim como em todos os problemas de gerenciamento, ter a equipe certa não garante o sucesso do projeto. Muitas outras coisas podem dar errado, incluindo mudanças nos negócios e no ambiente de negócios. No entanto, se você não prestar atenção à composição, à organização e às comunicações do grupo, você aumenta a probabilidade de que seu projeto seja executado com dificuldades.

22.3.1 Seleção de membros de grupo

O traballho do gerente ou líder de equipe é criar um grupo coeso e organizar seu grupo para que ele possa trabalhar com eficiência. Isso envolve a criação de um grupo com equilíbrio certo entre as habilidades técnicas e personalidades, assim como a organização desse grupo para que os membros trabalhem juntos com eficiência. Às vezes, pessoas de fora da organização são contratadas, no entanto, frequentemente, os grupos de engenharia de software são colocados juntos dos funcionários atuais com experiência em outros projetos. Raramente, contudo, os gerentes têm total liberdade na seleção da equipe. Muitas vezes, eles precisam usar as pessoas que estão disponíveis na empresa, mesmo quando estas não são as pessoas ideais para o trabalho.

Conforme discutido na Seção 22.2.1, muitos engenheiros de software são motivados, principalmente, por seu trabalho. Portanto, muitas vezes, os grupos de desenvolvimento de software são compostos de pessoas que têm suas próprias ideias a respeito de como os problemas técnicos devem ser resolvidos. Isso se reflete em problemas como ignorar os padrões de interface, sistemas sendo reprojetados enquanto são codificados, enfeites desnecessários para o sistema, entre outros.

Um grupo com personalidades complementares pode funcionar melhor do que um grupo selecionado unicamente pela capacidade técnica. Geralmente, as pessoas que são motivadas pelo trabalho são tecnicamente mais fortes. Provavelmente, as pessoas auto-orientadas se sairão melhor em incentivar o término do trabalho. As pessoas que são orientadas a interações ajudam a facilitar as comunicações dentro do grupo. Acredito ser particularmente importante ter em um grupo pessoas orientadas a interações. Elas gostam de conversar e podem detectar as tensões e divergências em um estágio inicial, antes que isso tenha um grande impacto sobre o grupo.

No estudo de caso do Quadro 22.3, eu sugeri como Alice, a gerente do projetos, tem tentado criar um grupo com personalidades complementares. Esse grupo tem uma boa mistura de pessoas orientadas a interações e a tarefas, mas, como discutido no Quadro 22.1, a personalidade auto-orientada de Dorothy causou problemas, pois ela não tem realizado o trabalho esperado. Fred, que trabalha meio período para o grupo como *expert* de domínio, também pode ser um problema. Ele se interessa por desafios técnicos, assim, pode não interagir bem

Quadro 22.3 Composição de grupo

Estudo de caso: Composição de grupo

Na criação de um grupo de desenvolvimento de tecnologia assistêncial, Alice está ciente da importância de selecionar membros com personalidades complementares. Ao entrevistar potenciais membros do grupo, ela tentou avaliar se eles são orientados a tarefas, auto-orientados ou orientados a interações. Ela sentiu que ela era, principalmente, um tipo auto-orientado, pois considerava o projeto como uma maneira de ser notada pela gerência sênior e, possivelmente, promovida. Portanto, para complementar a equipe, ela procurava por uma ou talvez duas personalidades orientadas a interações, além de indivíduos orientados a tarefas. Em sua última análise, a equipe que ela havia montado era:

Alice — auto-orientada
Brian — orientado a tarefas
Bob — orientado a tarefas
Carol — orientada a interações
Dorothy — auto-orientada
Ed — orientado a interações
Fred — orientado a tarefas

com os outros membros. O fato de que ele nem sempre faz parte da equipe indica que ele pode não se relacionar bem com os objetivos dela.

Às vezes, é impossível escolher um grupo com personalidades complementares. Se esse for o caso, o gerente de projeto precisa controlar o grupo para que suas metas individuais não tenham precedência sobre os objetivos organizacionais e os do grupo. Esse controle é mais fácil de obter se todos os membros do grupo participarem de todos os estágios do projeto. A iniciativa individual é mais provável quando aos membros do grupo são dadas instruções sem que eles estejam cientes do papel que sua tarefa tem no projeto geral.

Por exemplo, digamos que um engenheiro de software receba um projeto de programa para codificação e perceba o que parecem ser possíveis melhorias que poderiam ser feitas no projeto. Se ele ou ela implementasse essas melhorias sem compreender a lógica do projeto original, quaisquer mudanças, embora bem intencionadas, poderiam ter implicações adversas para outras partes do sistema. Se todos os membros do grupo estiverem envolvidos no projeto desde o início, eles entenderão por que essas decisões de projeto foram tomadas. Em seguida, eles podem identificar-se com essas decisões em vez de se oporem a elas.

22.3.2 Organização de grupo

A maneira como um grupo é organizado afeta as decisões tomadas por ele, a maneira como as informações são trocadas e as interações entre o grupo de desenvolvimento e os *stakeholders* externos. As questões organizacionais mais importantes para os gerentes de projetos são:

1. O gerente de projetos deve ser o líder técnico do grupo? O líder técnico ou arquiteto de sistema é responsável por decisões técnicas críticas tomadas durante o desenvolvimento de software. Às vezes, o gerente de projetos tem habilidade e experiência para assumir esse papel. No entanto, para grandes projetos, é melhor nomear um engenheiro sênior para ser o arquiteto de projeto, que assumirá a responsabilidade pela liderança técnica.
2. Quem será envolvido na tomada de decisões técnicas críticas e como estas serão tomadas? As decisões serão tomadas pelo arquiteto de sistema, pelo gerente de projetos ou por um consenso entre o maior número de membros de equipe?
3. Como serão tratadas as interações com os *stakeholders* externos e a gerência sênior da empresa? Em muitos casos, o gerente de projetos será responsável por essas interações, assistidas pelo arquiteto de sistema, caso este exista. No entanto, um modelo organizacional alternativo sugere a criação de um papel dedicado interessado nos contatos externos e a nomeação de alguém com habilidades de interação adequadas para esse papel.
4. Como os grupos podem integrar pessoas que não estão no mesmo local de trabalho? Atualmente, é comum os grupos incluírem membros de diferentes organizações e pessoas que trabalham em casa, bem como em escritórios compartilhados. Isso deve ser levado em consideração nos processos de tomada de decisão de grupo.
5. Como o conhecimento pode ser compartilhado entre o grupo? A organização de grupo afeta a troca de informações, assim como certos métodos de organização são melhores do que outros para o compartilhamento de informações. No entanto, você deve evitar muito compartilhamento de informações, pois as pessoas ficam sobrecarregadas e o excesso de informações pode distraí-las de seu trabalho.

Geralmente, pequenos grupos de programação são organizados de maneira bastante informal. O líder de grupo é envolvido no desenvolvimento de software com outros membros de grupo. Em um grupo informal, o trabalho a ser realizado é discutido pelo grupo como um todo e as tarefas são alocadas de acordo com habilidades e experiências. Os membros experientes do grupo podem ser responsáveis pelo projeto de arquitetura. No entanto, o projeto e implementação detalhada é de responsabilidade do membro da equipe que é alocado para uma determinada tarefa.

Os grupos de extreme programming (BECK, 2000) sempre são grupos informais. Entusiastas do XP afirmam que a estrutura formal inibe a troca de informações. Geralmente, em XP, muitas decisões percebidas como decisões de gestão (como as decisões de agendamento) são atribuídas aos membros de grupo. Os programadores trabalham juntos em pares para desenvolver o código e assumem responsabilidade conjunta sobre os programas desenvolvidos.

Os grupos informais podem ser muito bem-sucedidos, particularmente quando a maioria dos membros do grupo é experiente e competente. Esse grupo toma decisões por consenso, o que melhora o desempenho e a

coesão. No entanto, se a maior parte de um grupo for composta de membros inexperientes ou incompetentes, a informalidade poderá ser um obstáculo porque não há uma autoridade definitiva para direcionar o trabalho, causando falta de coordenação entre os membros de grupo e, possivelmente, eventuais falhas de projeto.

Grupos hierárquicos são grupos que têm uma estrutura hierárquica, na qual o líder do grupo está no alto da hierarquia. Ele ou ela tem mais autoridade do que os demais membros de grupo e, assim, pode direcionar o trabalho. Existe uma estrutura organizacional clara e as decisões são tomadas em direção ao topo da hierarquia e implementadas por pessoas mais abaixo nessa hierarquia. As comunicações são, principalmente, instruções de altos funcionários e existe relativamente pouca comunicação 'ascendente', desde os níveis inferiores até os superiores da hierarquia.

Essa abordagem pode funcionar bem quando um problema bem compreendido pode ser facilmente dividido em subproblemas, com as soluções para esses subproblemas sendo desenvolvidas em diferentes partes da hierarquia. Nessas situações, é necessária, relativamente, pouca comunicação em toda a hierarquia. No entanto, essas situações são relativamente raras, na engenharia de software, pelos seguintes motivos:

1. Muitas vezes, as mudanças no software requerem mudanças em várias partes do sistema, o que exige discussão e negociação em todos os níveis hierárquicos.
2. As tecnologias de software mudam tão rápido que, muitas vezes, o pessoal mais novo sabe mais sobre a tecnologia do que o pessoal mais experiente. Comunicações de cima para baixo (*top-down*) podem significar que o gerente de projeto não obtém informações sobre as possibilidades de uso de novas tecnologias. O pessoal mais novo pode tornar-se frustrado por causa do que vê como tecnologias ultrapassadas sendo usadas para desenvolvimento.

As organizações de grupos democráticos e hierárquicos não reconhecem, formalmente, que pode haver grandes diferenças na capacidade técnica entre os membros de grupo. Os melhores programadores podem ser até 25 vezes mais produtivos do que os piores programadores. Faz sentido usar as melhores pessoas de forma mais eficaz e fornecer-lhes o maior apoio possível. Um modelo organizacional inicial que foi concebido para dar esse apoio foi a equipe do programador-chefe.

Para fazer uso mais eficaz dos programadores altamente qualificados, Baker (1972) e outros (ARON, 1974; BROOKS, 1975) sugeriram que as equipes deviam ser construídas em torno de um programador-chefe individual, altamente qualificado. O princípio subjacente da equipe do programador-chefe é que o pessoal qualificado e experiente deve ser responsável por todo o desenvolvimento de software. Esse pessoal não deve se preocupar com questões de rotina e deve ter bom apoio técnico e administrativo para seu trabalho. Eles devem se concentrar no software a ser desenvolvido e não devem gastar muito tempo em reuniões externas.

No entanto, a organização da equipe do programador-chefe é, em minha opinião, muito dependente do próprio programador-chefe e seu assistente. Outros membros de equipe que não recebem responsabilidades suficientes podem tornar-se desmotivados porque sentem que suas habilidades são subutilizadas. Eles não têm informações para lidar caso as coisas andem mal e não recebem oportunidades de participar na tomada de decisões. Existem significativos riscos de projeto associados a essa organização de grupo, e estes podem superar quaisquer benefícios que esse tipo de organização possa trazer.

22.3.3 Comunicações de grupo

É absolutamente essencial que os membros de grupo se comuniquem de maneira eficaz e eficiente com outros membros e com os *stakeholders* de outros projetos. Os membros de grupo devem trocar informações sobre o *status* de seu trabalho, as decisões de projeto que foram feitas e as alterações nas decisões de projeto anteriores. Eles precisam resolver os problemas que surgem com outros *stakeholders* e informá-los das mudanças no sistema, no grupo e nos planos de entrega. Boa comunicação também ajuda a reforçar a coesão de grupo. Membros de grupo conseguem compreender as motivações, os pontos fortes e os pontos fracos das outras pessoas do grupo.

A eficácia e a eficiência das comunicações são influenciadas por:

1. *Tamanho de grupo*. Conforme um grupo se torna maior, a comunicação eficaz entre os membros fica mais difícil. O número de *links* de comunicação unidirecional é $n*(n-1)$, em que n é o tamanho do grupo, então, com um grupo de oito membros, existem 56 possíveis caminhos de comunicação. Isso significa que é possível que algumas pessoas raramente se comuniquem com outras. As diferenças de *status* entre os membros de grupo significam que as comunicações, frequentemente, são unidirecionais. Os gerentes e os engenheiros experientes tendem a dominar as comunicações com o pessoal menos experiente, que pode estar relutante em iniciar uma conversa ou fazer observações críticas.

2. *Estrutura de grupo.* Pessoas em grupos estruturados informalmente comunicam mais eficazmente do que as pessoas em grupos com uma estrutura formal, hierárquica. Em grupos hierárquicos, as comunicações tendem a subir e descer o fluxo da hierarquia. As pessoas do mesmo nível não podem conversar entre si. Esse é um problema ainda maior em projetos de grande porte com vários grupos de desenvolvimento. Se as pessoas que trabalham em diferentes subsistemas se comunicarem apenas por meio de seus gerentes, existe maior probabilidade de atrasos e mal-entendidos.
3. *Composição de grupo.* Pessoas com os mesmos tipos de personalidade (discutidos na Seção 22.2) podem colidir e, consequentemente, inibir as comunicações. Geralmente, a comunicação também é melhor em grupos mistos (MARSHALL e HESLIN, 1975) do que em grupos do mesmo sexo. Frequentemente, as mulheres são mais orientadas a interações do que os homens e podem atuar como controladoras e facilitadoras das interações para o grupo.
4. *Ambiente físico de trabalho.* A organização do local de trabalho é um fator importante para facilitar ou inibir as comunicações. Consulte o site de apoio do livro para obter mais informações: <www.pearson.com.br/sommerville>.
5. *Canais de comunicação disponíveis.* Existem muitas formas diferentes de comunicação: face a face, mensagens de correio electrônico, documentos formais, telefone e tecnologias da Web 2.0 como redes sociais e *wikis*. Como as equipes de projeto se tornam cada vez mais distribuídas, com membros de equipe trabalhando remotamente, você precisará fazer uso de várias tecnologias para facilitar a comunicação.

Geralmente, os gerentes de projetos trabalham com *deadlines* curtos e, consequentemente, eles podem tentar usar canais de comunicação que não tomem muito de seu tempo. Portanto, eles podem se basear em reuniões e documentos formais para passar informações para os *stakeholders* e para a equipe de projeto. Embora do ponto de vista do gerente de projeto isso possa ser uma abordagem eficiente para a comunicação, não costuma ser muito eficaz. Muitas vezes, existem boas razões para as pessoas não poderem assistir às reuniões e não ouvir a apresentação. Geralmente, os documentos longos não são lidos porque os leitores não sabem se eles são relevantes. Quando várias versões do mesmo documento são produzidas, os leitores encontram dificuldades em controlar as alterações.

Uma comunicação eficaz é alcançada quando a comunicação ocorre em dois sentidos e as pessoas envolvidas podem discutir problemas e informações e estabelecer um entendimento comum sobre as propostas e problemas. Isso pode ser feito por meio de reuniões, embora, frequentemente, estas sejam dominadas por personalidades mais fortes. Às vezes, é impraticável organizar reuniões a curto prazo. Cada vez mais, as equipes de projeto incluem membros remotos, o que também dificulta as reuniões.

Para combater esses problemas, você pode fazer uso de tecnologias da Web, como *wikis* e *blogs* para oferecer suporte à troca de informações. *Wikis* suportam a criação e edição colaborativa de documentos, e *blogs* suportam apenas simples discussões sobre questões e comentários feitos por membros de grupo. *Wikis* e *blogs* permitem que membros de projeto e *stakeholders* externos troquem informações, independentemente de sua localização. Tais mecanismos ajudam a gerenciar informações e manter o controle de segmentos de discussão, que muitas vezes se tornam confusos quando conduzidos por e-mail. Você também pode usar mensagens instantâneas e teleconferências, que podem ser facilmente organizadas, para resolver problemas que necessitem de discussão.

PONTOS IMPORTANTES

- Um bom gerenciamento de projetos de software é essencial, caso os projetos de engenharia de software devam ser desenvolvidos no prazo e dentro do orçamento.
- O gerenciamento de software é diferente de outros gerenciamentos de engenharia. O software é intangível. Projetos podem ser novos ou inovadores; assim, não existe um corpo de experiências para orientar seu gerenciamento. Os processos de software não são tão maduros quanto os processos de engenharia tradicionais.
- Atualmente, o gerenciamento de riscos é reconhecido como uma das mais importantes tarefas de gerenciamento de projetos.
- O gerenciamento de riscos envolve a identificação e a avaliação dos importantes riscos do projeto para estabelecer a probabilidade de que eles ocorram e as consequências para o projeto caso esses riscos ocorram. Você deve fazer planos para evitar, gerenciar ou lidar com riscos suscetíveis quando, e se, eles surgirem.

- As pessoas podem ser motivadas pela interação com outras pessoas, pelo reconhecimento da gerência e seus pares e pelas oportunidades de desenvolvimento pessoal.
- Grupos de desenvolvimento de software devem ser bastante pequenos e coesos. Os principais fatores que influenciam a eficácia de um grupo são as pessoas que o compõem, a forma como ele está organizado e a comunicação entre seus membros.
- As comunicações dentro de um grupo são influenciadas por fatores como o *status* dos membros do grupo, o tamanho do grupo, sua composição entre homens e mulheres, as personalidades e os canais de comunicação disponíveis.

The Mythical Man Month (Anniversary Edition). Os problemas de gerenciamento de software permanecem praticamente inalterados desde a década de 1960 e esse é um dos melhores livros sobre o tema. Um interessante e legível relato de gerenciamento de um dos primeiros grandes projetos de software, o sistema operacional IBM OS/360. A edição de aniversário (publicada 20 anos depois da edição original em 1975) inclui outros artigos clássicos de Brooks. (BROOKS, F. P. *The Mythical Man Month* (Anniversary Edition). Addison-Wesley, 1995.)

Software Project Survival Guide. Esse é um relato de gerenciamento de software, muito pragmático, que contém bons conselhos práticos, com um fundo de engenharia de software para os gerentes de projeto. É fácil de ler e compreender. (McCONNELL, S. *Software Project Survival Guide.* Microsoft Press, 1998.)

Peopleware: Productive Projects and Teams, 2nd edition. Essa é uma nova edição do livro clássico sobre a importância de tratar as pessoas corretamente ao se gerenciar projetos de software. É um dos poucos livros que reconhece a importância do lugar onde as pessoas trabalham. Altamente recomendado. (DeMARCO, T.; LISTER, T. *Peopleware: Productive Projects and Teams.* 2. ed. Dorset House, 1999.)

Waltzing with Bears: Managing Risk on Software Projects. Uma introdução muito prática e fácil de ler sobre riscos e gerenciamento de riscos. (DeMARCO, T.; LISTER, T. *Waltzing with Bears*: Managing Risk on Software Projects. Dorset House, 2003.)

22.1 Explique por que a intangibilidade dos sistemas de software gera problemas especiais para gerenciamento de projetos de software.

22.2 Explique por que os melhores programadores nem sempre se tornam os melhores gerentes de software. Você pode encontrar ajuda para basear sua resposta na lista de atividades de gerenciamento na Seção 22.1.

22.3 Usando instâncias de problemas de projetos relatados na literatura, liste as dificuldades e os erros de gerenciamento que ocorreram nesses projetos de programação que falharam. (Eu sugiro que você comece com *The Mythical Man Month*, por Fred Brooks.)

22.4 Além dos riscos mostrados na Tabela 22.1, identifique pelo menos seis outros possíveis riscos que podem surgir em projetos de software.

22.5 Contratos de preço fixo, em que o contratante oferece lances de um preço fixo para concluir um desenvolvimento de sistema, podem ser usados para mover o risco de projeto do cliente para o contratante. Se algo der errado, o contratante deve pagar. Sugira como o uso destes contratos pode aumentar a probabilidade de surgirem riscos de produtos.

22.6 Explique por que manter todos os membros de um grupo informados sobre progressos e decisões técnicas de um projeto pode melhorar a coesão de grupo.

22.7 Quais os problemas que podem surgir nas equipes de Extreme Programming em que muitas decisões de gerenciamento são atribuídas aos membros de equipe?

22.8 Escreva um estudo de caso, no estilo usado neste capítulo, para ilustrar a importância da comunicação em uma equipe de projeto. Suponha que alguns membros de equipe trabalhem remotamente e que não seja possível reunir toda a equipe a curto prazo.

22.9 Seu gerente solicitou que você entregue o software dentro do cronograma, o qual, você sabe, só poderá ser cumprido caso você peça a sua equipe de projeto que trabalhe horas extras não remuneradas. Todos os membros da equipe têm crianças pequenas. Discuta se você deve aceitar essa demanda de seu gerente ou se você deve convencer sua equipe a dar seu tempo para a organização, em vez de ficar com suas famílias. Quais fatores podem ser significativos em sua decisão?

22.10 Como programador, você tem a possibilidade de ser promovido para a posição de gerente de projetos, mas você sente que pode contribuir mais eficazmente como técnico ao invés de assumindo um papel gerencial. Discuta se você deve aceitar a promoção.

ARON, J. D. *The Program Development Process*. Reading, Mass.: Addison-Wesley, 1974.

BAKER, F. T. Chief Programmer Team Management of Production Programming. *IBM Systems J.*, v. 11, n. 1, 1972, p. 56-73.

BASS, B. M.; DUNTEMAN, G. Behaviour in groups as a function of self, interaction and task orientation. *J. Abnorm. Soc. Psychology*, v. 66, n. 4, 1963, p. 19-28.

BECK, K. *Extreme Programming Explained*. Reading, Mass.: Addison-Wesley, 2000.

BOEHM, B. W. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, v. 21, n. 5, 1988, p. 61-72.

BROOKS, F. P. *The Mythical Man Month*. Reading, Mass.: Addison-Wesley, 1975.

HALL, E. *Managing Risk: Methods for Software Systems Development*. Reading, Mass.: Addison-Wesley, 1998.

MARSHALL, J. E.; HESLIN, R. Boys and Girls Together. Sexual composition and the effect of density on group size and cohesiveness. *J. of Personality and Social Psychology*, v. 35, n. 5, 1975, p. 952-961.

MASLOW, A. A. *Motivation and Personality*. Nova York: Harper and Row, 1954.

OULD, M. *Managing Software Quality and Business Risk*. Chichesser: John Wiley & Sons, 1999.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 **23** 24 25 26

Planejamento de projeto

Objetivos

O objetivo deste capítulo é a introdução de planejamento de projetos, programação e estimativa de custos. Com a leitura deste capítulo, você:

- compreenderá os fundamentos de custos de softwares e as razões pelas quais o preço do software pode não ser diretamente relacionado a seu custo de desenvolvimento;
- saberá quais seções devem ser incluídas em um plano de projeto criado dentro de um processo de desenvolvimento dirigido a planos;
- compreenderá o que está envolvido na programação de projetos e o uso de gráficos de barras para apresentar um cronograma de projeto;
- conhecerá o 'jogo de planejamento', usado para dar suporte ao planejamento de projeto no Extreme Programming;
- entenderá como o modelo COCOMO II pode ser usado para a estimativa algorítmica de custos.

Conteúdo

Um dos trabalhos mais importantes de um gerente de projeto de software é o planejamento. Como gerente, você precisa desmembrar o trabalho em partes e atribuir essas partes aos membros de sua equipe; deve também antecipar os problemas que possam surgir e preparar soluções alternativas para eles. O plano de projeto, que é criado no início, é usado para comunicar à equipe e aos clientes como o trabalho será feito e para ajudar a avaliar o progresso do projeto.

No ciclo de vida do projeto, o planejamento ocorre em três estágios:

1. No estágio de proposta, quando você se propõe a fazer um contrato para desenvolver ou fornecer um sistema de software. Nesse estágio, você precisa de um plano para ajudá-lo a decidir se tem os recursos para concluir o trabalho e para calcular o preço que deve ser cotado para o cliente.
2. Durante a fase de iniciação de projeto, quando você precisa planejar quem vai trabalhar nele e como ele será dividido em incrementos, como os recursos serão alocados em toda sua empresa etc. Aqui, você tem mais informações do que no estágio de proposta e, portanto, pode refinar as estimativas de esforço iniciais, as quais você preparou.
3. Você modifica seu planejamento periodicamente, ao adquirir experiência e informações enquanto monitora a evolução do trabalho. Você aprende mais sobre o sistema que está sendo implementado e sobre a capacidade de sua equipe de desenvolvimento. Essa informação permite que você faça estimativas mais precisas de quanto tempo levará o trabalho. Além disso, os requisitos de software podem ser alterados, o que geralmente significa que a divisão

do trabalho precisa mudar e o cronograma precisa ser estendido. Para os projetos de desenvolvimento tradicionais, o plano elaborado durante a fase de iniciação precisa ser modificado. No entanto, quando é usada uma abordagem ágil, os planos são de curto prazo e mudam continuamente à medida que o software evolui. Na Seção 23.4 discuto o planejamento ágil.

O planejamento no estágio de proposta é inevitavelmente especulativo, pois não existe um conjunto completo de requisitos para o software que será desenvolvido. Em vez disso, você precisa responder a um convite para apresentação de propostas com base em uma descrição de alto nível da funcionalidade necessária para o software. Muitas vezes, um plano é requerido para uma proposta, assim, você precisa produzir um plano crível para a realização do trabalho. Normalmente, ao ganhar o contrato, é necessário replanejar o projeto, levando em consideração as alterações desde a realização da proposta.

Ao concorrer a um contrato, você precisa definir o preço que irá propor ao cliente para desenvolver o software. Como ponto de partida para o cálculo desse preço, você precisa elaborar uma estimativa de custos para completar o trabalho do projeto. A estimativa envolve quanto será o esforço necessário para concluir cada atividade e o custo total do trabalho. Você sempre deve calcular os custos de software objetivamente, a fim de prever com precisão o custo do desenvolvimento de software. Depois de ter uma estimativa razoável dos prováveis custos, você poderá calcular o preço para o cliente. Como discuto na próxima seção, muitos fatores influenciam o preço de um projeto de software — não é simplesmente custo + lucro.

Existem três parâmetros principais que você deve usar ao calcular os custos de um projeto de desenvolvimento de software:

- custos de esforço (os custos de pagamentos de gerentes e engenheiros de software);
- custos de hardware e software, incluindo manutenção;
- os custos de viagens e treinamentos.

Para a maioria dos projetos, o maior custo é o de esforço. Você deve estimar o esforço total (em pessoa-mês) que pode ser necessário para concluir o trabalho de um projeto. Obviamente, as informações para fazer tal estimativa são limitadas, assim, você precisa fazer a melhor estimativa possível e adicionar significativa contingência (tempo extra e esforço) no caso de sua estimativa inicial ser otimista.

Normalmente, para sistemas comerciais, usa-se hardwares de mercado, os quais são relativamente baratos. No entanto, os custos de software podem ser significativos se você precisar de licença para o *middleware* e o software de plataforma. Viagens longas podem ser necessárias quando um projeto é desenvolvido em locais diferentes. Embora os custos de viagem sejam uma pequena fração dos custos de esforço, o tempo de viagem é muitas vezes desperdiçado e aumenta significativamente os custos de esforço do projeto. Os sistemas eletrônicos de reuniões e outros softwares que oferecem suporte à colaboração remota podem reduzir a quantidade de viagens necessária. O tempo economizado pode ser dedicado mais ao trabalho produtivo do projeto.

Uma vez que você ganhou um contrato para desenvolver um sistema, o esboço do plano de projeto precisa ser refinado para criar um plano de iniciação. Nesse estágio, você deve saber mais sobre os requisitos para esse sistema. No entanto, pode não haver uma especificação completa de requisitos, especialmente se uma abordagem ágil para o desenvolvimento estiver sendo usada. Seu objetivo nesse estágio deve ser criar um plano de projeto que possa ser usado para oferecer suporte na tomada de decisões sobre a formação de equipe e elaboração do orçamento de projeto. Você usa o plano como uma base para a alocação de recursos para o projeto de dentro da organização e para ajudar a decidir se precisa contratar novos funcionários.

O plano também precisa definir mecanismos de monitoração de projeto. É necessário acompanhar o andamento do projeto comparando os custos e o progresso real e o planejado. Embora a maioria das organizações tenha procedimentos formais de monitoração, um bom gerente deve ser capaz de formar uma imagem clara do que está acontecendo por meio de discussões informais com a equipe de projeto. A monitoração informal pode prever potenciais problemas, revelando as dificuldades à medida que ocorrem. Por exemplo, discussões diárias com a equipe podem revelar um problema especial em encontrar um defeito de software. Em vez de aguardar o relato de atraso de cronograma, o gerente de projeto pode atribuir imediatamente um *expert* para o problema ou decidir a programação em torno dele.

O plano de projeto sempre evolui durante o processo de desenvolvimento. O planejamento de desenvolvimento destina-se a garantir que o plano de projeto continue a ser um documento útil para a equipe compreender o que está para ser alcançado e quando deverá ser entregue. Portanto, o cronograma, a estimativa de custos e os riscos precisam ser atualizados conforme o software é desenvolvido.

Se um método ágil é usado, existe ainda a necessidade de um plano de iniciação de projeto, independentemente da abordagem escolhida; a empresa ainda precisa planejar como os recursos para o projeto serão alocados. No entanto, esse não é um plano detalhado e deve incluir apenas informações limitadas sobre a divisão do trabalho e o cronograma de projeto. Durante o desenvolvimento, um plano de projeto e as estimativas de esforço informais são delineados para cada versão do software, com toda a equipe envolvida no processo de planejamento.

23.1 Definição de preço de software

Em princípio, o preço de um produto de software para o cliente é simplesmente o custo do desenvolvimento mais o lucro para o desenvolvedor. No entanto, na prática, o relacionamento entre o custo de projeto e o preço proposto para o cliente não costuma ser tão simples. Ao calcular um preço, você deve levar em consideração aspectos organizacionais, econômicos, políticos e comerciais, como aqueles mostrados na Tabela 23.1. Você precisa pensar sobre questões organizacionais, os riscos associados com o projeto e o tipo de contrato que será usado. Esses riscos podem alterar o preço para cima ou para baixo. Por causa das considerações organizacionais envolvidas, a decisão sobre o preço de projeto deve ser uma atividade de grupo envolvendo o pessoal de marketing e vendas, a gerência sênior e os gerentes de projeto.

Para ilustrar algumas das questões de definição de preço de projeto, considere o seguinte cenário:

Uma pequena empresa de software, PharmaSoft, emprega dez engenheiros de software. Ela acaba de concluir um projeto grande, mas só tem contratos que requerem cinco pessoas de desenvolvimento. No entanto, está concorrendo a um contrato muito grande, com uma grande empresa farmacêutica, que requer 30 pessoas/ano de esforço por dois anos. O projeto não será iniciado em pelo menos 12 meses, mas, caso ganhe, ele transformará as finanças da empresa.

A PharmaSoft recebe uma oportunidade de concorrer a um projeto que requer seis pessoas e precisa ser concluído em dez meses. Os custos (incluindo os overheads *desse projeto) são estimados em 1,2 milhão de dólares. No entanto, para melhorar sua posição competitiva, a empresa decide candidatar-se com o preço, para o cliente, de 800 mil dólares. Isso significa que, embora ela perca dinheiro no presente contrato, ela pode reter o pessoal especializado para projetos futuros mais rentáveis, que podem surgir no prazo de um ano.*

Como o custo de um projeto é fracamente relacionado ao preço indicado para o cliente, uma estratégia comumente usada é 'preço para ganhar'. Definir preço para ganhar significa que uma empresa tem uma ideia do preço que o cliente espera pagar e faz uma oferta para o contrato com base no preço esperado pelo cliente. Isso pode parecer antiético e incorreto, mas tem vantagens tanto para o cliente quanto para o provedor de sistema.

Tabela 23.1 Fatores que afetam a definição de preço de software

Fator	Descrição
Oportunidade de mercado	Uma organização de desenvolvimento pode cotar um preço baixo porque deseja mover-se para um novo segmento de mercado de software. Aceitar um baixo lucro em um projeto pode dar à organização a oportunidade de fazer um lucro maior no futuro. A experiência adquirida também pode ajudá-la a desenvolver novos produtos.
Incerteza de estimativa de custo	Se uma organização está insegura quanto a sua estimativa de custo, ela pode aumentar seu preço por uma contingência acima de seu lucro normal.
Condições contratuais	Um cliente pode estar disposto a permitir que o desenvolvedor mantenha a propriedade do código-fonte e o reutilize em outros projetos. O preço cobrado pode ser menor se o código-fonte do software for entregue ao cliente.
Volatilidade de requisitos	Se os requisitos podem ser alterados, uma organização pode reduzir seu preço para ganhar um contrato. Após a adjudicação, pode cobrar preços elevados por alterações nos requisitos.
Saúde financeira	Os desenvolvedores com dificuldades financeiras podem baixar seus preços para ganhar um contrato. É melhor obter um lucro menor do que sair do negócio. Em tempos economicamente difíceis, o fluxo de caixa é mais importante do que o lucro.

O custo de um projeto é acordado nas bases de uma proposta preliminar. Em seguida, as negociações acontecem entre cliente e consumidor, visando estabelecer a especificação detalhada de projeto. Essa especificação é restrita pelo custo acordado. O comprador e o vendedor devem concordar com a funcionalidade aceitável para o sistema. Em muitos projetos, os fatores fixos não são os requisitos do projeto, mas o custo. Para que o custo não seja excedido, os requisitos podem ser alterados.

Por exemplo, digamos que uma empresa (OilSoft) está oferecendo um contrato para desenvolver um sistema de entrega de combustível para uma companhia de petróleo que programa as entregas para seus postos de serviço. Não existe um documento de requisitos detalhados para esse sistema e a OilSoft estima que o valor de US$ 900 mil possa ser competitivo e estar dentro do orçamento da empresa de petróleo. Depois de concedido o contrato, a OilSoft negocia os requisitos detalhados do sistema para que a funcionalidade básica seja entregue. Eles então estimam os custos adicionais para outros requisitos. A companhia de petróleo não necessariamente perde, porque o contrato foi concedido a uma empresa confiável. Os requisitos adicionais podem ser financiados de um futuro orçamento, para que o orçamento da companhia petrolífera não seja perturbado por um custo inicial elevado de software.

23.2 Desenvolvimento dirigido a planos

Desenvolvimento dirigido a planos é uma abordagem da engenharia de software em que o processo de desenvolvimento é planejado em detalhes. Um plano de projeto é criado para registrar o trabalho a ser feito, quem o fará, o cronograma de desenvolvimento e os produtos do trabalho. Os gerentes usam o plano para suportar as tomadas de decisões de projeto e como uma forma de medir o progresso. O desenvolvimento dirigido a planos é baseado em técnicas de gerenciamento de projetos de engenharia e pode ser pensado como uma forma 'tradicional' de gerenciamento de grandes projetos de desenvolvimento de software. O que contrasta com o desenvolvimento ágil, no qual muitas decisões que afetam o desenvolvimento são postergadas e tomadas posteriormente, quando requeridas, durante o processo de desenvolvimento.

O principal argumento contra o desenvolvimento dirigido a planos são as decisões iniciais que precisam ser atualizadas por causa das mudanças no ambiente em que o software deve ser desenvolvido e usado. Postergar tais decisões é complicado, pois elas evitam retrabalho desnecessário. Os argumentos em favor de uma abordagem dirigida a planos são que o planejamento antecipado permite que questões organizacionais (disponibilidade de pessoal e outros projetos etc.) possam ser levadas em consideração e que dependências e problemas potenciais sejam descobertos antes do início do projeto, em vez de durante o andamento.

Em minha opinião, a melhor abordagem para o planejamento de projeto envolve uma mistura equilibrada entre o desenvolvimento baseado em planos e o ágil. O equilíbrio depende do tipo de projeto e das habilidades das pessoas que estão disponíveis. Em um extremo, sistemas críticos de proteção e segurança de grande porte requerem extensa análise inicial e podem precisar ser certificados antes de serem colocados em uso. Esses, principalmente, devem ser dirigidos a planos. No outro extremo, pequenos ou médios sistemas de informações, para serem usados em um ambiente competitivo que muda rapidamente, devem ser ágeis. Quando várias empresas estão envolvidas em um projeto de desenvolvimento, uma abordagem dirigida a planos é normalmente usada para coordenar o trabalho em cada área de desenvolvimento.

23.2.1 Planos de projeto

Em um projeto de desenvolvimento dirigido a planos, o plano de projeto define os recursos disponíveis para o projeto, a divisão de trabalho e o cronograma para a realização dos trabalhos. O plano deve identificar os riscos para o projeto e o software em desenvolvimento e a abordagem usada para o gerenciamento de riscos. Embora os detalhes específicos de planos de projeto variem dependendo do tipo do projeto e da organização, eles normalmente incluem as seguintes seções:

1. *Introdução*. Descreve brevemente os objetivos do projeto e define as restrições (por exemplo, orçamento, tempo etc.) que afetam o gerenciamento do projeto.
2. *Organização de projeto*. Descreve a maneira como a equipe de desenvolvimento é organizada, as pessoas envolvidas e seus papéis na equipe.
3. *Análise de riscos*. Descreve os possíveis riscos de projeto, a probabilidade desses riscos e as estratégias de redução de riscos propostas. O gerenciamento de risco é discutido no Capítulo 22.

4. *Requisitos de recursos de software e hardware.* Especifica o hardware e o suporte de software requerido para realizar o desenvolvimento. Se o hardware precisa ser comprado, as estimativas de preço e o cronograma de entrega podem ser incluídos.
5. *Divisão de trabalho.* Estabelece a partição do projeto em atividades e identifica os *milestones* e os resultados associados a cada atividade. Os *milestones* são estágios importantes do projeto, nos quais o progresso pode ser avaliado; os resultados são produtos de trabalho entregues ao cliente.
6. *Cronograma de projeto.* Mostra as dependências entre as atividades, a estimativa de tempo necessário para chegar a cada *milestone* e a alocação das pessoas para as atividades. As diferentes maneiras de apresentar o cronograma são discutidas na próxima seção do capítulo.
7. *Mecanismos de monitoração e geração de relatório.* Definem os relatórios de gerenciamento que devem ser produzidos, quando devem ser produzidos e os mecanismos de monitoramento de projetos que serão usados.

Assim como o plano principal de projeto, o qual deve se concentrar nos riscos para os projetos e para o cronograma, você pode desenvolver uma série de planos complementares para dar suporte a outras atividades de processo, como gerenciamento de configuração e testes. Exemplos de possíveis planos complementares são mostrados na Tabela 23.2.

23.2.2 O processo de planejamento

O planejamento de projetos é um processo iterativo que começa quando um plano inicial de projeto é criado, na fase de iniciação. A Figura 23.1 traz um diagrama de atividades da UML que mostra um *workflow* típico para o processo de planejamento de projeto. Mudanças nos planos são inevitáveis. Conforme mais informações sobre o sistema e a equipe de projeto se tornam disponíveis durante o projeto, você deve atualizar regularmente o plano para refletir as mudanças de requisitos, de cronograma e de riscos. As mudanças de metas de negócios também levam a mudanças nos planos de projeto. As metas de negócios mudam e podem afetar todos os projetos, os quais podem precisar de um novo planejamento.

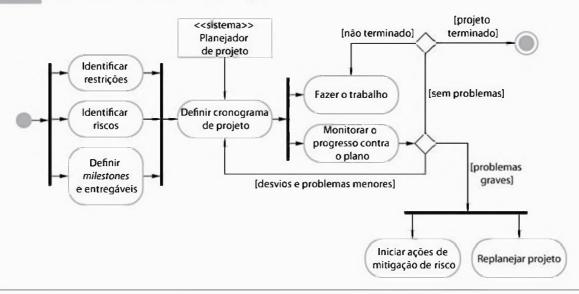
No início de um processo de planejamento, você deve avaliar as restrições que afetam o projeto. Essas restrições são a data de entrega requerida, o pessoal disponível, o orçamento global, as ferramentas disponíveis e assim por diante. Junto com isso, você também deve identificar os *milestones* e os entregáveis do projeto. *Milestones* são pontos no cronograma pelos quais você pode avaliar o progresso, por exemplo, a entrega do sistema para testes. Os entregáveis são produtos de trabalho que são entregues ao cliente (por exemplo, um documento de requisitos para o sistema).

Em seguida, o processo entra em um *loop*. Você elabora um cronograma estimado para o projeto e as atividades definidas no cronograma são iniciadas ou recebem permissão para continuar. Após um tempo (normalmente cerca de duas a três semanas), você deve revisar os progressos e observar as discrepâncias com o cronograma previsto. Como as estimativas iniciais de parâmetros de projeto são inevitavelmente aproximadas, pequenos desvios são normais e modificações no plano original terão de ser feitas.

É muito importante que você seja realista quando estiver criando o plano de um projeto. Durante um projeto, quase sempre surgem problemas com alguma descrição e eles podem levar a atrasos. Os pressupostos iniciais e o cronograma devem ser pessimistas. Em seu plano, deve haver contingência suficiente para que os *milestones* e as restrições de projeto não precisem ser renegociados cada vez que se aproximem do *loop* de planejamento.

Tabela 23.2 Suplementos de plano de projeto

Plano	Descrição
Plano de qualidade	Descreve os procedimentos de qualidade e as normas que serão usadas em um projeto.
Plano de validação	Descreve a abordagem, os recursos e o cronograma usados para validação de sistema.
Plano de gerenciamento de configuração	Descreve os procedimentos e as estruturas de gerenciamento de configuração que serão usados.
Plano de manutenção	Prevê os requisitos de manutenção, custos e esforço.
Plano de desenvolvimento de pessoal	Descreve como as habilidades e experiências dos membros de equipe do projeto serão desenvolvidas.

Figura 23.1 O processo de planejamento de projeto

Se houver problemas graves com o trabalho de desenvolvimento, os quais podem causar atrasos significativos, você precisará iniciar ações de mitigação de risco para reduzir os riscos de falha de projeto. Em conjunto com essas ações, também será preciso replanejar o projeto, podendo envolver a renegociação das restrições e dos entregáveis de projeto com o cliente. Também precisará ser estabelecido e acordado com o cliente um novo cronograma para quando o trabalho deve ser concluído.

Caso essa renegociação não tenha êxito ou as ações de mitigação de risco sejam ineficazes, você deve se organizar para uma revisão técnica formal de projeto. Os objetivos dessa revisão são encontrar uma abordagem alternativa que permita a continuidade do projeto e verifique se o projeto e os objetivos do cliente e do desenvolvedor do software ainda estão alinhados.

O resultado de uma revisão pode ser a decisão de cancelar um projeto. Pode ser o resultado de falhas técnicas ou gerenciais, mas, frequentemente, é uma consequência de mudanças externas que afetam o projeto. O tempo de desenvolvimento de um grande projeto de software costuma abranger vários anos. Durante esse tempo, os objetivos do negócio e as prioridades inevitavelmente se alteram. Essas mudanças podem significar que o software não é mais necessário ou que os requisitos originais de projeto são inadequados. A gerência pode decidir suspender o desenvolvimento de software ou fazer grandes mudanças no projeto para refletir as mudanças nos objetivos organizacionais.

23.3 Programação de projeto

A programação de projeto é o processo de decidir como será organizado o trabalho de um projeto em tarefas separadas e quando e como essas tarefas serão executadas. Você estima o tempo de calendário e o esforço necessários para concluir cada tarefa, bem como quem vai trabalhar nas tarefas identificadas. Você também deve estimar os recursos necessários para concluir cada tarefa, como o espaço em disco requerido em um servidor, o tempo requerido de um hardware especializado, tal qual um simulador, e qual será o orçamento de viagens. Em termos de estágios de planejamento, discutidos na introdução deste capítulo, durante a fase de iniciação de projeto geralmente se cria um cronograma inicial de projeto. Esse cronograma é refinado e modificado durante o planejamento do desenvolvimento.

Ambos os processos — os baseados em planos e os ágeis — precisam de um cronograma inicial de projeto, embora o nível de detalhes possa ser menor em um plano de projeto ágil. Esse cronograma inicial é usado para planejar como as pessoas serão alocadas para os projetos e para verificar o progresso do projeto analisando seus compromissos contratuais. Nos processos tradicionais de desenvolvimento, inicialmente, o cronograma completo é desenvolvido e modificado conforme o andamento do projeto. Em processos ágeis, deve haver um cronograma

global que identifique quando as principais fases do projeto serão concluídas. Uma abordagem iterativa para a programação é usada para planejar cada fase.

A programação de projetos dirigidos a planos (Figura 23.2) envolve quebrar o trabalho total de um projeto em tarefas separadas e estimar o tempo necessário para a conclusão de cada tarefa. Normalmente, as tarefas devem durar pelo menos uma semana e não mais do que dois meses. Uma subdivisão mais correta indica que uma quantidade desproporcional de tempo deve ser gasta no replanejamento e na atualização do plano de projeto. O tempo aproximado máximo para qualquer tarefa deve ser de oito a dez semanas. Se demorar mais do que isso, a tarefa deve ser subdividida por planejamento e programação de projeto.

Algumas dessas tarefas são realizadas em paralelo, com diferentes pessoas trabalhando em diferentes componentes do sistema. Você precisa coordenar essas tarefas paralelas e organizar o trabalho para que a força de trabalho seja usada corretamente e não introduzir dependências desnecessárias entre as tarefas. É importante evitar uma situação em que todo o projeto esteja atrasado porque uma tarefa crítica está inacabada.

Se um projeto é tecnicamente avançado, certamente as estimativas iniciais serão otimistas, mesmo quando você tenta considerar todas as eventualidades. Nesse sentido, o cronograma de software não é diferente do cronograma de qualquer outro grande projeto avançado. Novas aeronaves, pontes e mesmo novos modelos de carros estão frequentemente atrasados por causa de problemas imprevistos. Portanto, os cronogramas devem ser continuamente atualizados à medida que melhores informações sobre o progresso se tornam disponíveis. Se o projeto que está sendo programado for semelhante a um projeto anterior, as estimativas anteriores poderão ser reutilizadas. No entanto, os projetos podem usar diferentes métodos de projeto e linguagens de implementação, de modo que a experiência em projetos anteriores pode não ser aplicável ao planejamento de um novo projeto.

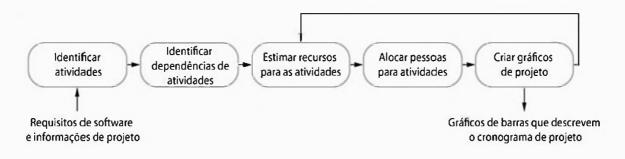
Ao estimar os cronogramas, você deve considerar a possibilidade de que as coisas deem errado. As pessoas que estão trabalhando em um projeto podem ficar doentes ou sair, o hardware pode falhar e o software ou hardware de apoio essencial pode ser entregue com atraso. Se o projeto for novo e tecnicamente avançado, partes dele poderão vir a ser mais difíceis e demorar mais tempo do que o inicialmente previsto.

Uma boa regra é estimar como se nada fosse dar errado e aumentar sua estimativa para cobrir os problemas previstos. Um fator de contingência para cobrir os problemas imprevistos também pode ser adicionado na estimativa. Esse fator de contingência extra depende do tipo de projeto, dos parâmetros de processo (*deadlines*, normas etc.) e da qualidade e experiência dos engenheiros de software que trabalham no projeto. As estimativas de contingência podem adicionar de 30 a 50% para o esforço e o tempo necessários para o projeto.

23.3.1 Representação de cronograma

Os cronogramas do projeto podem ser representados em uma tabela ou em uma planilha mostrando as tarefas, o esforço, a duração esperada e as dependências de tarefas (Tabela 23.3). No entanto, esse estilo de representação dificulta a visualização dos relacionamentos e das dependências entre as diferentes atividades. Por isso, foram desenvolvidas alternativas de representações gráficas de cronogramas de projetos, as quais muitas vezes são mais fáceis de serem lidas e compreendidas. Existem dois tipos de representações comumente usadas:

1. Gráficos de barras, que são baseados em calendário e mostram quem é responsável por cada atividade, o tempo decorrido previsto e quais atividades estão programadas para começar e terminar. Os gráficos de barras são chamados 'gráficos de Gantt', em homenagem a seu inventor, Henry Gantt.

Figura 23.2 O processo de programação de projeto

2. Redes de atividades, que são diagramas de rede e mostram as dependências entre as diferentes atividades que compõem o projeto.

Normalmente, uma ferramenta de planejamento de projeto é usada para gerenciar informações a respeito do cronograma de projeto. Essas ferramentas esperam que você insira informações sobre o projeto em uma tabela e, em seguida, crie um banco de dados com informações sobre ele. Os gráficos de barras e gráficos de atividades podem ser gerados automaticamente a partir desse banco de dados.

As atividades de projeto são o elemento básico de planejamento. Cada atividade tem:

1. Duração de dias ou meses.
2. Estimativa de esforço, que reflete o número de pessoas/dia ou pessoas/mês para concluir o trabalho.
3. *Deadline* para a conclusão da atividade.
4. Ponto de término definido, que representa o resultado tangível de término da atividade. Este poderia ser um documento, ou a realização de uma reunião de revisão, ou a execução bem-sucedida de todos os testes etc.

Ao planejar um projeto, você também deve definir *milestones*; ou seja, cada estágio do projeto em que pode ser feita uma avaliação de progresso. Cada *milestone* deve ser documentado por um relatório curto do qual conste o resumo dos progressos e dos trabalhos realizados. Os *milestones* podem ser associados com uma única tarefa ou com os grupos de atividades relacionadas. Por exemplo, na Tabela 23.3, o *milestone* M1 está associado com a tarefa T1, e o *milestone* M3, com um par de tarefas, T2 e T4.

Um tipo especial de *milestone* é a produção de um entregável de projeto. Um entregável é o produto de trabalho que será entregue ao cliente. É o resultado de uma fase significativa do projeto, assim como a especificação ou o projeto. Geralmente, os entregáveis são especificados no contrato do projeto e a percepção do cliente sobre o andamento do projeto depende desses entregáveis.

Para ilustrar como os gráficos de barras são usados, criei um conjunto hipotético de tarefas, como mostrado na Tabela 23.3. Essa tabela exibe as tarefas, o esforço estimado, a duração e as interdependências de tarefas. Nela, você percebe que a tarefa T3 é dependente da tarefa T1. A tarefa T1 deve, portanto, ser concluída antes de começar a T3. Por exemplo, a T1 pode ser a preparação de um projeto de componente e a T3, a implementação desse projeto. Antes de iniciar a implementação, o projeto deve estar completo. Observe que a duração estimada para algumas tarefas é maior do que o esforço necessário e vice-versa. Se o esforço for menor que a duração, significa que as pessoas alocadas a essa tarefa não estão trabalhando em tempo integral. Se o esforço exceder a duração, significa que vários membros da equipe estão trabalhando ao mesmo tempo na tarefa.

Tabela 23.3 Tarefas, durações e dependências

Tarefa	Esforço (pessoas-dias)	Duração (dias)	Dependências
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

A Figura 23.3 usa as informações da Tabela 23.3 e apresenta o cronograma de projeto em um gráfico. Trata-se de um gráfico de barras que mostra um calendário de projeto, assim como as datas iniciais e finais das tarefas. A leitura da esquerda para a direita mostra claramente, no gráfico de barras, quando as tarefas começam e terminam. Os *milestones* (M1, M2 etc.) também são exibidos no gráfico de barras. Observe que as tarefas independentes são realizadas em paralelo (por exemplo, as tarefas T1, T2 e T4 começam no início do projeto).

Assim como planejar o cronograma de entrega para o software, os gerentes de projeto precisam alocar os recursos para as tarefas. O principal recurso, obviamente, são os engenheiros de software que realizarão o trabalho, os quais precisam ser alocados para as atividades de projeto. A alocação de recursos também pode ser entrada para as ferramentas de gerenciamento de projeto, gerando um gráfico de barras que mostra quando os funcionários estão trabalhando no projeto (Figura 23.4). As pessoas podem estar trabalhando em mais de uma tarefa ao mesmo tempo e, às vezes, elas não estão trabalhando no projeto. Elas podem estar de férias, trabalhando em outros projetos, em cursos de treinamento, ou participando de outra atividade. Eu mostro as alocações de meio período, usando uma linha diagonal que cruza a barra.

Geralmente, as organizações de grande porte empregam vários especialistas para trabalharem em um projeto. Na Figura 23.4, você pode ver que Mary é uma especialista e que trabalha apenas em uma tarefa no projeto. Isso pode causar problemas no cronograma. Se um projeto atrasa enquanto um especialista está trabalhando nele, isso pode ter um efeito dominó sobre os outros projetos em que o especialista também trabalha. Esses projetos podem ser adiados porque o especialista, no caso, Mary, não está disponível.

Se uma tarefa está atrasada, isso certamente pode afetar as últimas tarefas que dependem dela. Estas não podem começar até que seja a tarefa atrasada concluída. Atrasos podem causar sérios problemas com alocação de pessoal, especialmente quando as pessoas estão trabalhando em vários projetos ao mesmo tempo. Se uma tarefa (T) está atrasada, a equipe responsável por essa tarefa pode ser remanejada para outro trabalho (W). Terminar esse trabalho pode demorar mais do que o atraso, mas, uma vez remanejados, eles não podem ser realocados para a tarefa original, T. Isso pode causar mais atrasos em T enquanto eles completam W.

Figura 23.3 Gráfico de barras de atividades

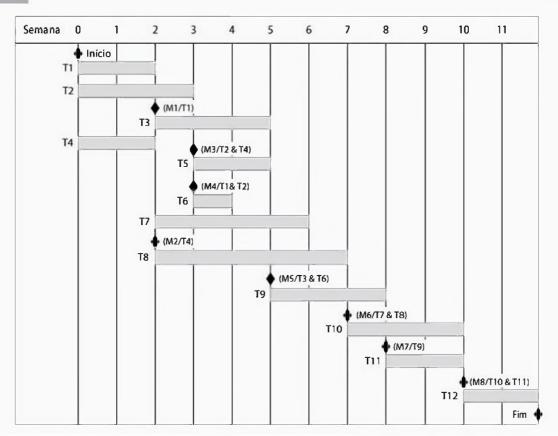
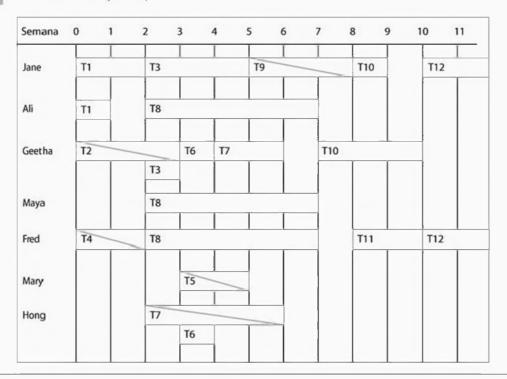



Figura 23.4 Gráfico de alocação de pessoal

23.4 Planejamento ágil

Os métodos ágeis de desenvolvimento de software são as abordagens iterativas nas quais o software é desenvolvido e entregue aos clientes em incrementos. Ao contrário das abordagens dirigidas a planos, a funcionalidade desses incrementos não é planejada com antecedência, mas decidida durante o desenvolvimento. A decisão sobre o que incluir em um incremento depende do progresso e das prioridades do cliente. O argumento para essa abordagem é que as prioridades e as necessidades do cliente mudam e, assim, faz sentido ter um plano flexível capaz de acomodar essas mudanças. O livro de Cohn (COHN, 2005, p. 1735) mostra uma discussão abrangente sobre questões de planejamento em projetos ágeis.

As abordagens ágeis mais comumente usadas, como Scrum (SCHWABER, 2004) e Extreme Programming (BECK, 2000), têm uma abordagem de planejamento em dois estágios, correspondentes ao estágio de iniciação no desenvolvimento dirigido a planos e planejamento de desenvolvimento:

1. Planejamento de *release*, que olha à frente por vários meses e decide sobre as características que devem ser incluídas em um *release* de um sistema.
2. Planejamento de iteração, que tem uma perspectiva de curto prazo e foca o planejamento do próximo incremento de um sistema. Normalmente, são duas a quatro semanas de trabalho para a equipe.

No Capítulo 3 a abordagem Scrum para o planejamento foi discutida, por isso me concentro agora em planejamento no Extreme Programming (XP). Esse é chamado de 'jogo de planejamento' (Figura 23.5) e geralmente envolve toda a equipe de desenvolvimento, incluindo os representantes do cliente.

Figura 23.5 Planejamento em XP

A especificação de sistema em XP baseia-se em histórias de usuários que refletem as características que devem ser incluídas no sistema. No início do projeto, a equipe e o cliente tentam identificar um conjunto de histórias, que abrange toda funcionalidade que será incluída no sistema final. Inevitavelmente, alguma funcionalidade estará ausente, mas, nesse estágio, isso não é importante.

O próximo estágio é de estimativas. A equipe do projeto lê e discute as histórias e as classifica de acordo com o tempo que acredita ser necessário para implementá-las. Isso pode envolver transformar histórias grandes em histórias menores. Muitas vezes, a estimativa relativa é mais fácil do que a estimativa absoluta. Geralmente, as pessoas acham difícil estimar a quantidade de esforço ou o tempo necessário para fazer algo. No entanto, quando são apresentadas com várias coisas para fazer, elas podem julgar quais histórias levarão mais tempo e maior esforço. Uma vez terminada a classificação, a equipe aloca pontos imaginários de esforço para as histórias. Uma história complexa pode ter oito pontos e uma história simples, dois pontos. Isso deve ser feito para todas as histórias na lista classificada.

Uma vez que as histórias tenham sido estimadas, o esforço relativo é traduzido na primeira estimativa de esforço total necessário, usando a noção de 'velocidade'. Em XP, a velocidade é o número de pontos de esforço implementados pela equipe, por dia. Isso pode ser estimado a partir da experiência anterior ou por meio do desenvolvimento de uma ou duas histórias para ver quanto tempo é necessário. A estimativa de velocidade é aproximada, mas é refinada durante o processo de desenvolvimento. Uma vez que se tenha uma estimativa de velocidade, é possível calcular o esforço total em pessoa-dia necessário para se implementar o sistema.

O planejamento de *release* envolve a seleção e o refinamento das histórias que refletirão os recursos que serão implementados em um *release* de um sistema e a ordem em que as histórias devem ser implementadas. O cliente precisa estar envolvido nesse processo. É escolhida uma data de *release* e as histórias são examinadas para analisar se a estimativa de esforço é consistente com essa data. Se não, algumas histórias devem ser adicionadas ou removidas da lista.

O planejamento de iteração é o primeiro estágio do processo do desenvolvimento de iteração. As histórias a serem implementadas para essa iteração são escolhidas, com o número de histórias que refletem o tempo para entregar uma iteração (normalmente, duas ou três semanas) e a velocidade da equipe. Quando chega a data de entrega de uma iteração, ela deve estar completa, mesmo que todas as histórias não tenham sido implementadas. A equipe considera as histórias que foram implementadas e adiciona seus pontos de esforço. Assim, a velocidade pode ser recalculada e usada no planejamento do próximo release do sistema.

No início de cada iteração, há um estágio de planejamento mais detalhado em que os desenvolvedores quebram as histórias em tarefas de desenvolvimento. Uma tarefa de desenvolvimento deve levar de 4 a 16 horas. Todas as tarefas devem ser concluídas para implementar todas as histórias listadas naquela iteração. Em seguida, os desenvolvedores individuais inscrevem-se para as tarefas específicas que implementarão. Cada desenvolvedor conhece sua velocidade individual e não deve candidatar-se para mais tarefas do que pode implementar.

Existem dois benefícios importantes dessa abordagem para alocação de tarefas:

1. Toda a equipe obtém uma visão geral das tarefas a serem concluídas em uma iteração. Portanto, compreendem o que os outros membros de equipe estão fazendo e com quem devem falar caso sejam identificadas dependências entre as tarefas.
2. Desenvolvedores individuais escolhem as tarefas a implementar. Eles não recebem as tarefas do gerente de projeto, portanto, existe um sentimento de propriedade que provavelmente os motivará a concluir a tarefa.

Na metade do caminho de uma iteração, o progresso é avaliado. Nesse estágio, a metade dos pontos de esforço de história deve ser concluída. Assim, se uma iteração envolve 24 pontos de história e 36 tarefas, então 12 pontos de história e 18 tarefas devem ser concluídos. Se esse não for o caso, o cliente deve ser consultado e algumas histórias devem ser removidas da iteração.

Essa abordagem do planejamento tem a vantagem de o software sempre ser liberado no tempo previsto e não ocorrer nenhum deslize no cronograma. Se o trabalho não puder ser concluído dentro do prato estabelecido, a filosofia XP é reduzir o escopo do trabalho, em vez de estender o cronograma. No entanto, em alguns casos, o incremento pode não ser suficiente para ser útil. Reduzir o escopo pode gerar trabalho extra para os clientes caso eles precisem usar um sistema incompleto ou mudar suas práticas de trabalho entre um *release* do sistema e outro.

Uma grande dificuldade em planejamento ágil é a dependência da disponibilidade e do envolvimento do cliente. Na prática, isso pode ser difícil de organizar, pois, às vezes, o representante do cliente deve priorizar outros trabalhos. Os clientes podem estar mais familiarizados com planos de projetos tradicionais e podem ter dificuldades em se envolver em um projeto de planejamento ágil.

O planejamento ágil funciona bem com equipes de desenvolvimento pequenas e estáveis, as quais podem reunir-se e discutir as histórias que serão implementadas. No entanto, quando as equipes são grandes e/ou distribuídas

geograficamente, ou quando membros de equipe mudam com frequência, é praticamente impossível que todos possam ser envolvidos no planejamento colaborativo, essencial para o gerenciamento de projeto ágil. Como consequência, os grandes projetos são geralmente planejados usando-se métodos tradicionais de gerenciamento de projetos.

23.5 Técnicas de estimativa

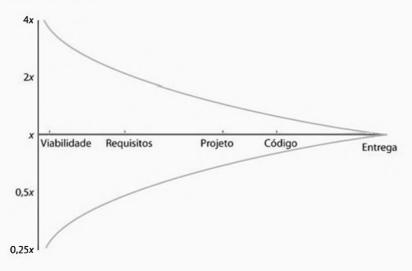
As estimativas de cronograma de projeto são difíceis. Talvez você precise fazer estimativas iniciais de uma definição de requisitos de usuário de alto nível. O software pode ser executado em computadores desconhecidos ou usar novas tecnologias de desenvolvimento. Provavelmente, as pessoas envolvidas no projeto e suas habilidades não serão conhecidas. Existem tantas incertezas que é impossível estimar com precisão os custos de desenvolvimento de sistema durante os estágios iniciais de um projeto.

Existe até uma dificuldade fundamental na avaliação da precisão das diferentes abordagens para a estimativa de custo e esforço. Geralmente, as estimativas de projeto são de autopreenchimento. A estimativa é usada para definir o orçamento de projeto e o produto é ajustado para que o orçamento seja realizado. Um projeto que está dentro do orçamento pode ter alcançado isso em detrimento dos recursos do software que está sendo desenvolvido.

Não conheço qualquer experimento controlado com custos de projeto em que os custos estimados não tenham sido usados para influenciar o experimento. Um experimento controlado não revelaria a estimativa de custo para o gerente de projeto. Os custos reais, em seguida, seriam comparados aos custos estimados de projeto. No entanto, as organizações precisam fazer estimativas de esforço e custo de software. Existem dois tipos de técnicas que podem ser usadas para isso:

1. *Técnicas baseadas em experiências*. As estimativas de requisitos de futuros esforços se baseiam na experiência do gerente em projetos anteriores e em seu domínio de aplicação. Essencialmente, o gerente faz uma avaliação informada do que os requisitos de esforço podem ser.
2. *Modelagem algorítmica de custos*. Uma abordagem usada em muitas situações para calcular o esforço de projeto com base em estimativas de atributos de produto, como tamanho e características do processo, por exemplo a experiência do pessoal envolvido.

Em ambos os casos, você precisará usar seu julgamento para estimar o esforço, ou estimar as características de projeto e de produto. Na fase de iniciação de um projeto, essas estimativas têm uma grande margem de erro. Com base em dados coletados de um grande número de projetos, Boehm e colegas (1995) descobriram que as estimativas iniciais variam significativamente. Se a estimativa inicial de esforço necessário é de x meses de esforço, o intervalo pode ser desde $0{,}25x$ até $4x$ do esforço real medido quando o sistema for entregue. Durante o planejamento de desenvolvimento, as estimativas tornam-se cada vez mais precisas à medida que o projeto avança (Figura 23.6).

Figura 23.6 Incerteza de estimativa

As técnicas baseadas em experiência contam com as experiências anteriores do gerente de projetos e o esforço real dispendido nesses projetos em atividades relacionadas ao desenvolvimento de software. Normalmente, você identifica os entregáveis que devem ser produzidos em um projeto e os diferentes componentes de software ou sistemas que serão desenvolvidos. Você os documenta em uma planilha, estima cada um individualmente e calcula o esforço total necessário. Geralmente, é útil contar com um grupo de pessoas envolvidas na estimativa de esforço e pedir a cada membro do grupo para explicar sua estimativa. Isso muitas vezes revela fatores que outros não levaram em consideração. Em seguida, você itera a uma estimativa acordada pelo grupo.

A dificuldade com as técnicas baseadas em experiências é que um novo projeto de software pode não ter muito em comum com projetos anteriores. O desenvolvimento de software muda muito rápida e frequentemente; um projeto usará técnicas desconhecidas como *web services*, desenvolvimento baseado em COTS ou AJAX. Se você não trabalhou com essas técnicas, sua experiência anterior não pode ajudá-lo a estimar o esforço necessário, tornando mais difícil produzir os custos precisos e estimativas de cronograma.

23.5.1 Modelagem algorítmica de custos

A modelagem algorítmica de custos usa uma fórmula matemática para prever os custos do projeto com base em estimativas de tamanho de projeto, tipo de software que está sendo desenvolvido e outros fatores de equipe, processo e produto. Um modelo algorítmico de custos pode ser construído analisando-se os custos e os atributos de projetos concluídos e encontrando-se a fórmula que melhor se ajuste à experiência real.

Modelos algorítmicos de custos são usados principalmente para fazer estimativas dos custos de desenvolvimento de software. No entanto, Boehm e colegas (2000) discutem uma variedade de outros usos para esses modelos, como a elaboração de estimativas para investidores em empresas de software, estratégias alternativas para ajudar a avaliar os riscos e para decisões informadas sobre reúso, redesenvolvimento ou terceirização.

Os modelos algorítmicos para estimar o esforço em um projeto de software baseiam-se principalmente em uma fórmula simples:

$$\text{Esforço} = A \times \text{Tamanho}^{B} \times M$$

`A` é um fator constante que depende das práticas organizacionais locais e do tipo de software em desenvolvimento. `Tamanho` pode ser uma avaliação do tamanho do código do software ou uma estimativa da funcionalidade expressa em pontos de função ou de aplicação. O valor do expoente `B` encontra-se geralmente entre 1 e 1,5. `M` é um multiplicador feito pela combinação de atributos de processo, produtos e de desenvolvimento, como os requisitos de confiança para o software e a experiência da equipe de desenvolvimento.

O número de linhas de código-fonte (SLOC, do inglês *source lines of code*) no sistema entregue é a métrica fundamental de tamanho usada em muitos modelos algoritmos de custos. A estimativa do tamanho pode envolver as estimativas por analogia com outros projetos, a estimativa por conversão de pontos de função ou aplicação para o tamanho de código e a estimativa pela classificação dos tamanhos de componentes do sistema e pelo uso de componente de referência conhecido para estimar o tamanho de componente, ou pode ser simplesmente uma questão de julgamento de engenharia.

A maioria dos modelos algorítmicos de estimativa tem um componente exponencial (`B` na equação anterior) que está relacionado ao tamanho e à complexidade do sistema, o qual reflete o fato de, geralmente, os custos não aumentarem linearmente com o tamanho do projeto. À medida que o tamanho e a complexidade do software aumentam, ocorrem custos extras por causa do *overhead* de comunicação entre as equipes maiores, do gerenciamento de configuração mais complexo, da integração mais difícil do sistema e assim por diante. Quanto mais complexo o sistema, mais esses fatores afetam o custo. Portanto, o valor de `B` geralmente aumenta com o tamanho e a complexidade do sistema.

Todos os modelos algorítmicos têm problemas semelhantes:

1. Muitas vezes, é difícil estimar o `Tamanho` em um estágio de um projeto, quando apenas a especificação está disponível. Estimativas de ponto de função e ponto de aplicação (ver adiante) são mais fáceis de se produzir do que as estimativas do tamanho de código, mas, às vezes, ainda são imprecisas.
2. As estimativas dos fatores que contribuem para `B` e `M` são subjetivas. As estimativas variam de uma pessoa para outra, dependendo de sua formação e experiência com o tipo de sistema que está sendo desenvolvido.

Estimar o tamanho exato de código é difícil no estágio inicial de um projeto porque o tamanho do programa final depende de decisões de projeto que não podem ter sido tomadas no momento em que a estimativa está sendo requisitada. Por exemplo, uma aplicação que requer o gerenciamento de dados de alto desempenho pode implementar seu próprio sistema de gerenciamento de dados ou usar um sistema comercial de banco de dados. Na estimativa inicial de custo, é improvável saber se existe um sistema de banco de dados comercial que funcione bem o suficiente para satisfazer os requisitos de desempenho. Portanto, não se sabe quanto código de gerenciamento de dados será incluído no sistema.

A linguagem de programação usada para o desenvolvimento de sistemas também afeta o número de linhas de código que devem ser desenvolvidas. Uma linguagem como Java pode indicar que mais linhas de código são necessárias do que se C (por exemplo) fosse usada. No entanto, esse código extra permite uma maior verificação em tempo de compilação, de maneira que os custos de validação sejam reduzidos. Como isso deve ser considerado? Além disso, é possível reusar uma quantidade significativa de código de projetos anteriores e a estimativa de tamanho precisa ser ajustada para considerar essa possibilidade.

Modelos algorítmicos de custos são uma maneira sistemática de estimar o esforço necessário para desenvolver um sistema. No entanto, esses modelos são complexos e difíceis de usar. Existem muitos atributos e uma margem considerável de incerteza em estimar esses valores. Essa complexidade desencoraja potenciais usuários e, portanto, a aplicação prática da modelagem algorítmica de custos foi limitada a um pequeno número de empresas.

Outra barreira para o uso de modelos algorítmicos é a necessidade de calibração. Os usuários desses modelos devem calibrá-los e atribuir a eles valores usando seus próprios dados históricos de projeto, pois estes refletem experiências e práticas locais. No entanto, poucas organizações coletaram dados suficientes de projetos antigos em um formulário que suporte a calibração de modelo. No entanto, o uso prático dos modelos algorítmicos deve começar com os valores publicados para os parâmetros de modelo. É praticamente impossível para um modelador saber o quanto eles se relacionam com sua organização.

Se você usar um modelo algorítmico de estimativa de custos, deve desenvolver uma série de estimativas (piores, esperadas e melhores), ao invés de uma única estimativa, e aplicar a fórmula de custo para todas elas. É mais fácil as estimativas serem precisas quando você entende o tipo de software que está sendo desenvolvido, tenha calibrado o modelo de custo usando dados locais, ou quando opções de hardware e linguagem de programação são predefinidas.

23.5.2 O modelo COCOMO II

Vários modelos similares têm sido propostos para ajudar a estimar o esforço, o cronograma e os custos de um projeto de software. O modelo que discuto aqui é o COCOMO II. Trata-se de um modelo empírico, obtido por meio da coleta de dados de um grande número de projetos de software. Esses dados foram analisados para descobrir as fórmulas que melhor se ajustam às observações. Essas fórmulas vinculam o tamanho do sistema e o produto, os projetos e os fatores de equipe ao esforço para desenvolver o sistema. O modelo COCOMO II é um modelo de estimativa bem documentado e não proprietário.

O modelo COCOMO II foi desenvolvido a partir de um modelo COCOMO anterior de estimativas de custo, o qual foi em grande parte baseado no desenvolvimento do código original (BOEHM, 1981; BOEHM e ROYCE, 1989). O modelo COCOMO II leva em consideração abordagens mais modernas para o desenvolvimento de software, tais como desenvolvimento rápido usando linguagens dinâmicas, desenvolvimento por composição de componentes e uso de programação de banco de dados. O COCOMO II suporta o modelo de desenvolvimento em espiral, descrito no Capítulo 2, e incorpora submodelos que produzem estimativas altamente detalhadas.

Os submodelos (Figura 23.7) que fazem parte do modelo COCOMO II são:

1. *Um modelo de composição de aplicações*. Ele modela o esforço necessário para desenvolver sistemas criados a partir de componentes reusáveis, *scripts* ou programação de banco de dados. As estimativas de tamanho de software baseiam-se em pontos de aplicação e em uma fórmula simples tamanho/produtividade usada para estimar o esforço necessário. O número de pontos de aplicação em um programa é uma estimativa do número de telas separadas que são exibidas, o número de relatórios que são produzidos, o número de módulos em linguagens de programação imperativas (como o Java) e o número de linhas de linguagem *script* ou código de programação de banco de dados.

Figura 23.7 Modelos de estimativa COCOMO

Número de pontos de aplicação — Baseado em — Modelo de composição de aplicação — Usado para — Sistemas desenvovidos usando linguagens dinâmicas, programação DB etc.

Número de pontos de função — Baseado em — Modelo de projeto preliminar — Usado para — Estimativa de esforço inicial baseada nos requisitos de sistema e em opções de projeto

Número de linhas de código reusado ou gerado — Baseado em — Modelo de reúso — Usado para — Esforço para integrar componentes reusáveis ou código gerado automaticamente

Número de linhas de código-fonte — Baseado em — Modelo de pós-arquitetura — Usado para — Esforço de desenvolvimento baseado em especificações de projeto de sistema

2. *Um modelo de projeto preliminar*. Esse modelo é usado durante as fases iniciais de projeto do sistema após os requisitos serem estabelecidos. A estimativa é baseada na fórmula-padrão de estimativa abordada na introdução, com um conjunto simplificado de sete multiplicadores. As estimativas são baseadas nos pontos de função, os quais são convertidos em número de linhas de código-fonte. Os pontos de função são uma forma independente de linguagem de quantificar a funcionalidade do programa. Você calcula o número total de pontos de função em um programa medindo ou estimando o número total de entradas e saídas externas, as interações de usuário, as interfaces externas e os arquivos ou tabelas de banco de dados usados pelo sistema.
3. *Um modelo de reúso*. Esse modelo é usado para calcular o esforço necessário para integrar os componentes reusáveis e/ou os códigos de programa gerados automaticamente. Normalmente, é usado em conjunto com o modelo de pós-arquitetura.
4. *Um modelo de pós-arquitetura*. Uma vez que a arquitetura de sistema tenha sido projetada, uma estimativa mais precisa do tamanho de software pode ser feita. Novamente, esse modelo usa a fórmula-padrão de estimativa de custo discutida anteriormente. No entanto, ele inclui um conjunto mais amplo de 17 multiplicadores, refletindo a capacidade pessoal, o produto e as características de projeto.

Naturalmente, em grandes sistemas, diferentes partes do sistema podem ser desenvolvidas usando-se diferentes tecnologias, e você não precisa estimar todas as partes do sistema com o mesmo nível de precisão. Em tais casos, pode-se usar o submodelo apropriado para cada parte do sistema e combinar os resultados para criar uma estimativa composta.

O modelo de composição de aplicação

O modelo de composição de aplicação foi introduzido em COCOMO II para apoiar os esforços de estimativa necessários para prototipação de projetos e para projetos em que o software é desenvolvido pela composição de componentes existentes. É baseado na estimativa de aplicação ponderada de pontos (chamados de pontos de objeto), divididos por um padrão estimado de produtividade de ponto de aplicação. Assim, a estimativa é ajustada de acordo com a dificuldade de desenvolvimento de cada ponto de aplicação (BOEHM et al., 2000). A produtividade depende da experiência e da capacidade do desenvolvedor, assim como dos recursos das ferramentas de software (ICASE) usadas para apoiar o desenvolvimento. A Tabela 23.4 mostra os níveis de produtividade de ponto de aplicação sugeridos pelos desenvolvedores de COCOMO (BOEHM et al., 1995).

Geralmente, a composição de aplicação envolve um significativo reúso de software. É quase certo que alguns pontos de aplicação no sistema serão implementados usando componentes reusáveis. Consequentemente, você precisa ajustar a estimativa para levar em consideração a porcentagem de reúso esperada. Portanto, a fórmula final para cálculo do esforço em protótipos de sistema será:

Tabela 23.4 Produtividade de pontos de aplicação

Experiência e capacidade do desenvolvedor	Muito baixa	Baixa	Nominal	Alta	Muito alta
Capacidade e maturidade ICASE	Muito baixa	Baixa	Nominal	Alta	Muito alta
PROD (NAP/mês)	4	7	13	25	50

```
PM = (NAP × (1 - %reúso/100)) / PROD
```

PM é a estimativa de esforço em pessoa-mês. NAP é o número total de pontos de aplicação do sistema entregue. "%reúso" é uma estimativa da quantidade de código reusado no desenvolvimento. PROD é a produtividade de pontos de aplicação, como mostrado na Tabela 23.4. O modelo produz uma estimativa aproximada, pois não leva em consideração os esforços adicionais envolvidos no reúso.

O modelo de projeto preliminar

Esse modelo pode ser usado durante os estágios iniciais de um projeto, antes do projeto de arquitetura detalhado para o sistema estar disponível. As primeiras estimativas de projeto são mais úteis para explorar opções quando você precisa comparar diferentes maneiras de implementar os requisitos de usuários. O modelo de projeto preliminar assume que os requisitos de usuários foram acordados e que os estágios iniciais do processo de projeto de sistema estão em curso. Nesse estágio, sua meta deve ser uma estimativa de custo rápida e aproximada. Portanto, você deve levantar hipóteses simplificadoras, por exemplo, de que o esforço envolvido na integração de código reusável é zero.

As estimativas produzidas nesse estágio são basedas em uma fórmula-padrão para modelos algorítmicos, que é:

$$\text{Esforço} = A \times \text{Tamanho}^{B} \times M$$

Baseado em seu grande conjunto de dados, Boehm propôs que o coeficiente A deveria ser 2,94. O tamanho do sistema é expresso em KSLOC (do inglês *Kilo Source Lines Of Code*), o qual é o número de milhares de linhas de código-fonte. Você pode calcular o KSLOC estimando o número de pontos de função no software. Em seguida, você usa tabelas-padrão que se relacionam com o tamanho do software para pontos de função para diferentes linguagens de programação, a fim de calcular as estimativas iniciais do tamanho do sistema em KSLOC.

O expoente B reflete o esforço aumentado necessário na medida em que aumenta o tamanho do projeto. Este pode variar de 1,1 para 1,24 dependendo da novidade do projeto, da flexibilidade de desenvolvimento, os processos de resolução de risco usados, da coesão da equipe de desenvolvimento e do nível de maturidade do processo (ver Capítulo 26) da organização. Eu discuto como o valor desse expoente é calculado usando esses parâmetros na descrição do modelo de pós-arquitetura de COCOMO II.

Isso resulta em um cálculo de esforço, como segue:

$$PM = 2{,}94 \times \text{Tamanho}^{(1.1\ -\ 1.24)} \times M$$

em que

$$M = PERS \times RCPX \times RUSE \times PDIF \times PREX \times FCIL \times SCED$$

O multiplicador M é baseado em sete atributos de projeto e processo que aumentam ou diminuem a estimativa. Os atributos usados no modelo de projeto preliminar são a confiabilidade e complexidade de produto (RCPX), reúso requerido (RUSE), dificuldade de plataforma (PDIF), capacidade de pessoal (PERS), experiência de pessoal (PREX), cronograma (SCED) e recursos de apoio (FCIL). Esses atributos são explicados no site de apoio do livro (<www.pearson.com.br/sommerville>). Os valores para esses atributos devem ser estimados usando uma escala de seis pontos, em que 1 corresponde a 'muito baixo' e 6 corresponde a 'muito alto'.

O modelo de reúso

Como discutido no Capítulo 16, o reúso de software é comum atualmente. A maioria dos grandes sistemas inclui uma quantidade significativa de códigos que foram usados em projetos anteriores de desenvolvimento. O modelo de reúso é usado para estimar o esforço necessário para integrar códigos reusáveis ou gerados.

O COCOMO II considera dois tipos de códigos reusáveis. O código 'Caixa Preta' é um código que pode ser reusado sem compreensão ou alterações. O esforço de desenvolvimento para o código de caixa preta é igual a zero. O código 'Caixa Branca' precisa ser adaptado para se integrar com o novo código ou outros componentes reusáveis. O esforço de desenvolvimento é requerido no reúso, porque os códigos precisam ser entendidos e modificados para poderem funcionar corretamente no sistema.

Muitos sistemas incluem códigos automaticamente gerados a partir de modelos de sistema, como discutido no Capítulo 5. Um modelo (frequentemente em UML) é analisado e gera-se um código para implementar os objetos especificados no modelo. O modelo de reúso COCOMO II inclui uma fórmula para estimar o esforço necessário para integrar o código gerado:

$$PM_{Auto} = (ASLOC \times AT/100) / ATPROD \quad // \text{ Estimativa para código gerado}$$

ASLOC é o número total de linhas de código reusado, incluindo o código que é gerado automaticamente.

AT é a porcentagem de códigos reusados gerados automaticamente.

ATPROD é a produtividade dos engenheiros na integração do código.

Boehm et al. (2000) mediram ATPROD para ser cerca de 2.400 declarações-fonte por mês. Portanto, se existe um total de 20 mil linhas de código-fonte reusado em um sistema e 30% deste é gerado automaticamente, o esforço necessário para integrar o código gerado é:

$$(20.000 \times 30/100) / 2.400 = 2.5 \text{ pessoas - mês} \quad // \text{ Código gerado}$$

Um cálculo de esforço separado é usado para estimar o esforço necessário para integrar o código reusado de outros sistemas. O modelo de reúso não calcula o esforço diretamente a partir de uma estimativa do número de componentes reusáveis. Em vez disso, baseado no número de linhas de códigos que são reusados, o modelo fornece uma base para calcular o número equivalente de linhas de código novo (ESLOC). Este é baseado no número de linhas de código reusável que deve ser alterado e um multiplicador que reflete a quantidade de trabalho necessária para reusar os componentes. A fórmula para calcular ESLOC leva em consideração o esforço necessário para entender o software, fazer alterações no código reusado e alterar o sistema para integração do código.

A fórmula seguinte é usada para calcular o número de linhas equivalentes do código-fonte:

$$ESLOC = ASLOC \times AAM$$

ESLOC é o número de linhas equivalentes do novo código-fonte.

ASLOC é o número de linhas de código nos componentes que precisam ser alterados.

AAM é um 'multiplicador de ajuste de adaptação', conforme discutido a seguir.

Reúso nunca é gratuito, e existem custos mesmo se o reúso não se mostrar viável. No entanto, os custos de reúso diminuem conforme aumenta o total de códigos reusados. Os custos fixos de compreensão e avaliação são espalhados por mais linhas de código. O multiplicador de ajuste de adaptação (AAM) ajusta a estimativa para refletir o esforço adicional necessário para reúso de código. Simplificando, o AAM é a soma de três componentes:

1. Um componente de adaptação (referido como AAF) que representa os custos de alterar o código reusado. O componente de adaptação inclui subcomponentes que levam em consideração alterações de projeto, código e integração.
2. Um elemento de compreensão (referido como SU) que representa os custos de compreender o código para ser reusado e a familiaridade do engenheiro com esse código. SU varia de cinquenta para códigos não estruturados complexos e dez para códigos bem escritos, orientados a objetos.
3. Um fator de avaliação (referido como AA) que representa os custos de tomada de decisão de reúso. Ou seja, algumas análises sempre devem decidir quando o código pode ser reusado e isso é incluído nos custos como AA. AA varia de zero a oito, dependendo da quantidade de esforço de análise necessários.

Caso alguma adaptação de código possa ser feita automaticamente, o esforço necessário diminui. Portanto, você ajusta a estimativa calculando o percentual de código automaticamente adaptado (AT) e usando isso para ajustar ASLOC. Portanto, a fórmula final é:

$$ESLOC = ASLOC \times (1 - AT/100) \times AAM$$

Uma vez que o ESLOC tenha sido calculado, você deve aplicar a fórmula-padrão de estimativa para calcular o esforço total necessário, no qual o parâmetro Tamanho = ESLOC. Em seguida, você adiciona esse resultado ao esforço para integrar automaticamente o código gerado, que já foi calculado, computando o esforço total necessário.

O modelo de pós-arquitetura

O modelo de pós-arquitetura é o mais detalhado dos modelos COCOMO II. Ele é usado uma vez que um projeto de arquitetura preliminar esteja disponível para o sistema de maneira que a estrutura do subsistema seja conhecida. Então, é possível fazer estimativas para cada parte do sistema.

O ponto de partida para as estimativas produzidas no modelo de pós-arquitetura é a mesma fórmula básica usada nas primeiras estimativas de design:

$$PM = A \times Tamanho^{B} \times M$$

Nesse estágio do processo, você deve ser capaz de fazer uma estimativa mais precisa do tamanho do projeto, pois sabe como o sistema será decomposto em objetos ou módulos. Você faz essa estimativa do tamanho do código usando três parâmetros:

1. Uma estimativa do número total de linhas de código novo para ser desenvolvido (SLOC).
2. Uma estimativa dos custos de reúso baseado em um número equivalente de linhas de código-fonte calculado (ESLOC), usando o modelo de reúso.
3. Uma estimativa do número de linhas de código que podem ser modificadas por causa de alterações nos requisitos de sistema.

Você adiciona os valores desses parâmetros para calcular o tamanho total de código, em KSLOC, que é usado na fórmula de cálculo de esforços. O componente final na estimativa — o número de linhas de código modificado — reflete o fato de que sempre existem mudanças nos requisitos de software. Isso gera retrabalho e desenvolvimento de código extra, o qual você deve levar em consideração. Naturalmente, é comum haver mais incertezas nesse cálculo do que nas estimativas de código novo a ser desenvolvido.

O termo expoente (B) na fórmula de cálculo de esforços está relacionado com os níveis de complexidade de projeto. À medida que os projetos se tornam mais complexos, os efeitos do aumento do tamanho de sistema se tornam mais significativos. No entanto, os procedimentos e as boas práticas organizacionais podem controlar essa 'deseconomia' de escala, que é uma consequência da crescente complexidade. Portanto, conforme mostrado na Tabela 23.5, o valor do expoente B baseia-se em cinco fatores. Esses fatores são classificados em uma escala de seis pontos de 0 a 5, em que 0 significa 'extra alto' e 5 significa 'muito baixo'. Para calcular B, você adiciona as classificações, divide por 100 e adiciona o resultado a 1,01 para obter o expoente que deve ser usado.

Por exemplo, imagine que uma organização vá assumir um projeto em um domínio em que tem pouca experiência. O cliente do projeto ainda não definiu o processo que será usado ou o tempo previsto no cronograma de projeto para análise de riscos significativos. Uma nova equipe de desenvolvimento deve ser formada para implementar esse sistema. A organização recentemente pôs em prática um programa de melhoria de processo e foi classificada como uma organização de Nível 2 de acordo com a avaliação de capacidade SEI, conforme discutido no Capítulo 26. Os valores possíveis para as classificações utilizadas no cálculo de expoente são:

Tabela 23.5 Fatores de escala usados no cálculo de expoente no modelo de pós-arquitetura

Fator de escala	Explicação
Precedência	Reflete a experiência anterior da organização com esse tipo de projeto. Muito baixa significa nenhuma experiência anterior; muito alta significa que a organização está completamente familiarizada com esse domínio de aplicação.
Flexibilidade de desenvolvimento	Reflete o grau de flexibilidade no processo de desenvolvimento. Muito baixa significa que um processo predeterminado é usado; muito alta significa que o cliente define apenas metas gerais.
Arquitetura/Resolução de riscos	Reflete a medida da análise de risco realizada. Muito baixa significa pouca análise; muito alta significa uma análise de riscos completa e exaustiva.
Coesão de equipe	Reflete quão bem os membros da equipe de desenvolvimento conhecem uns aos outros e trabalham juntos. Muito baixa significa muita dificuldade de interação; muito alta significa uma equipe integrada e eficaz, sem problemas de comunicação.
Maturidade de processo	Reflete a maturidade do processo da organização. O cálculo desse valor depende do questionário de maturidade do CMM, mas uma estimativa pode ser alcançada subtraindo-se o nível de maturidade de processo CMM de 5.

1. *Precedência*, pontuação baixa (4). Trata-se de um novo projeto para a organização.
2. *Flexibilidade de desenvolvimento*, pontuação muito alta (1). Nenhum envolvimento do cliente no processo de desenvolvimento, de maneira que existem algumas alterações impostas externamente.
3. *Arquitetura/Resolução de riscos*, pontuação muito baixa (5). Nenhuma análise de risco foi realizada.
4. *Coesão da equipe*, pontuação nominal (3). Trata-se de uma nova equipe, então não há informação disponível sobre a coesão.
5. *Maturidade do processo*, pontuação nominal (3). Algum controle de processo existe.

A soma desses valores é 16. Em seguida, você calcula o expoente dividindo-o por 100 e adicionando-o a 0,01. O valor ajustado de B é 1,17.

A estimativa de esforço geral é refinada usando um extenso conjunto de 17 atributos de produto, processo e organização (*drivers* de custo), em vez dos sete atributos usados no modelo de projeto preliminar. Você pode estimar os valores para esses atributos, pois tem mais informações sobre o software propriamente dito, seus requisitos não funcionais, a equipe e o processo de desenvolvimento.

A Tabela 23.6 mostra como os atributos de *driver* de custo podem influenciar as estimativas de esforço. Assumi um valor de 1,17 para o expoente, conforme discutido no exemplo anterior, e assumi que RELY, CPLX, STOR, TOOL e SCED são os *drivers* de custo mais importantes do projeto. Todos os outros *drivers* de custo têm um valor nominal de 1, de forma que não afetam o cálculo do esforço.

Na Tabela 23.6, eu assumi valores máximos e mínimos para os *drivers* de custo mais importantes, para mostrar como eles influenciam a estimativa de esforço. Os valores são aqueles do manual de referência do COCOMO II (BOEHM, 2000). Você pode ver que os valores altos para os *drivers* de custo geram uma estimativa de esforço três vezes maior que a estimativa inicial, considerando que valores baixos reduzem em média a estimativa para um terço do original. Isso destaca as diferenças significativas entre diferentes tipos de projetos e as dificuldades em transferir a experiência de um domínio de aplicação para outro.

23.5.3 Duração de projeto e seleção de pessoal

Ao estimar o custo geral de um projeto e o esforço necessário para desenvolver um sistema de software, os gerentes de projeto também devem estimar quanto tempo o software levará para ser desenvolvido, além de quantas pessoas serão necessárias para trabalhar no projeto. Cada vez mais, as empresas estão exigindo cronogramas de desenvolvimento mais curtos para que seus produtos possam ser introduzidos no mercado antes do produto do concorrente.

Tabela 23.6 O efeito dos *drivers* de custo nas estimativas de esforço

Valor de expoente	1,17
Tamanho de sistema (incluindo fatores para reúso e volatilidade de requisitos)	128.000 DSI
Estimativa inicial de COCOMO sem *drivers* de custo	**730 pessoas-mês**
Confiabilidade	Muito alta, multiplicador = 1,39
Complexidade	Muito alta, multiplicador = 1,3
Restrição de memória	Alta, multiplicador = 1,21
Uso de ferramentas	Baixo, multiplicador = 1,12
Cronograma	Acelerado, multiplicador = 1,29
Estimativa de COCOMO ajustada	**2.306 pessoas-mês**
Confiabilidade	Muito baixa, multiplicador = 0,75
Complexidade	Muito baixa, multiplicador = 0,75
Restrição de memória	Nenhum, multiplicador = 1
Uso de ferramentas	Muito alto, multiplicador = 0,72
Cronograma	Normal, multiplicador = 1
Estimativa de COCOMO ajustada	**295 pessoas-mês**

O modelo COCOMO inclui uma fórmula para estimar o tempo necessário para conclusão de um projeto:

$$TDEV = 3 \times (PM)^{(0.33 + 0.2*(B - 1.01))}$$

TDEV é o cronograma nominal para o projeto, em meses de calendário, ignorando qualquer multiplicador relacionado com o cronograma de projeto.

PM é o esforço calculado pelo modelo COCOMO.

B é o expoente relacionado com a complexidade, conforme discutido na Seção 23.5.2.

Se B = 1,17 e PM = 60, então,

$$TDEV = 3 \times (60)^{0.36} = 13 \text{ meses}$$

No entanto, o cronograma de projeto nominal previsto pelo modelo COCOMO e o cronograma requerido pelo plano de projeto não são necessariamente a mesma coisa. Pode haver um requisito para entregar o software mais cedo ou (mais raramente) após a data sugerida pelo cronograma nominal. Se o cronograma deve ser comprimido, o esforço necessário para o projeto deve ser aumentado. Esse cálculo é considerado pelo multiplicador SCED no cálculo de estimativa de esforço.

Suponha que um projeto estime o TDEV em 13 meses, tal como sugerido anteriormente, mas o cronograma real requerido seja de 11 meses. Isso representa uma compressão no cronograma de cerca de 25%. Usando os valores para o multiplicador SCED obtidos pela equipe de Boehm, o multiplicador de esforço para tal compressão de cronograma é 1,43. Portanto, o esforço real necessário para respeitar esse cronograma acelerado é quase 50% maior do que o esforço necessário para entregar o software de acordo com o cronograma nominal.

Existe um relacionamento complexo entre o número de pessoas trabalhando em um projeto, o esforço que será dedicado ao projeto e o cronograma de entrega de projeto. Se quatro pessoas podem concluir um projeto em 13 meses (ou seja, 52 pessoas-mês de esforço), então você pode pensar que, adicionando mais uma pessoa, você pode concluir o trabalho em 11 meses (55 pessoas-mês de esforço). No entanto, o modelo COCOMO sugere que, na verdade, você precisará de mais pessoas para concluir o trabalho em 11 meses (66 pessoas-mês de esforço).

A razão para isso é que adicionar pessoas de fato reduz a produtividade dos membros de equipe existente e, assim, o incremento real do esforço adicionado é inferior a uma pessoa. Conforme a equipe de projeto aumenta de tamanho, os membros de equipe gastam mais tempo se comunicando e definindo interfaces entre as partes do sistema desenvolvido por outras pessoas. Portanto, a duplicação do número de pessoas (por exemplo) não significa que a duração do projeto será reduzida pela metade. Se a equipe de desenvolvimento for grande, poderá acontecer de, ao serem adicionadas mais pessoas, o cronograma de desenvolvimento se estender, em vez de diminuir. Myers (1989) discute os problemas de aceleração de cronograma. Ele sugere que os projetos podem ter problemas significativos caso tentem desenvolver o software sem dar tempo suficiente para a conclusão do trabalho.

Você não pode simplesmente estimar o número de pessoas necessárias para a equipe de um projeto dividindo o esforço total pelo cronograma de projeto necessário. Normalmente, no início de um projeto, um pequeno número de pessoas é necessário para realizar o projeto inicial. Durante o desenvolvimento e os testes do sistema, a equipe cresce até seu máximo e, em seguida, diminui de tamanho, enquanto o sistema está sendo preparado para a implantação. Um crescimento muito rápido do número de pessoas na equipe de projeto tem sido relacionado com atrasos no cronograma de projeto. Portanto, os gerentes do projeto devem evitar adicionar pessoal demais em um projeto no início de sua vida útil.

Esse acúmulo de esforço pode ser modelado pelo que é chamado de uma curva de Rayleigh (LONDEIX, 1987). O modelo de estimativa de Putnam (1978), que incorpora um modelo de seleção de pessoal de projeto, baseia-se nessas curvas de Rayleigh. Esse modelo também inclui o tempo de desenvolvimento como um fator-chave. À medida que o tempo de desenvolvimento é reduzido, o esforço necessário para o desenvolvimento do sistema cresce exponencialmente.

- O preço cobrado por um sistema não depende apenas dos custos estimados de desenvolvimento e do lucro exigido pela empresa de desenvolvimento. Fatores organizacionais podem significar que o preço seja aumentado para compensar riscos altos, ou reduzido para ganhar vantagem competitiva.
- Muitas vezes, o preço do software é definido para ganhar um contrato e, em seguida, a funcionalidade do sistema é ajustada para atender ao preço estimado.

- O desenvolvimento dirigido a planos é organizado em torno de um plano completo de projeto, que define as atividades de projeto, o esforço planejado, o cronograma de atividades e quem é responsável por cada atividade.
- A definição de cronograma de projeto envolve a criação de várias representações gráficas de parte do plano de projeto. Os gráficos de barras, que mostram a duração da atividade e as linhas de tempo de pessoal, são as representações de cronograma mais usadas.
- Um *milestone* de projeto é o resultado previsível de uma atividade ou um conjunto de atividades. Em cada *milestone* de projeto, um relatório formal de progresso deve ser apresentado para a gerência. Um entregável é o produto de trabalho, o qual é entregue ao cliente de projeto.
- O jogo de planejamento do XP envolve toda a equipe no planejamento de projeto. O plano é desenvolvido de forma incremental e, se houver problemas, ele é ajustado para que a funcionalidade de software seja reduzida de maneira a não atrasar a entrega de um incremento.
- As técnicas de estimativa de software podem ser baseadas em experiências, em que os gerentes analisam o esforço necessário, ou algorítmico, no qual o esforço necessário é calculado a partir de outros parâmetros de projetos estimados.
- O modelo de custos COCOMO II é um modelo algorítmico de custos maduro, que, ao formular uma estimativa de custos, leva em consideração os atributos de projeto, produto, hardware e pessoal.

LEITURA COMPLEMENTAR

Software Cost Estimation with COCOMO II. Livro definitivo sobre o modelo COCOMO II. Fornece uma descrição completa do modelo com muitos exemplos e inclui o software que implementa o modelo. Ele é extremamente detalhado e de uma leitura avançada. (BOEHM B. et al. *Software Cost Estimation with COCOMO II.* Prentice Hall, 2000.)

'Ten unmyths of project estimation'. Artigo pragmático que aborda as dificuldades práticas de estimativa de projetos e desafia alguns pressupostos fundamentais nessa área. (ARMOUR, P. *Comm. ACM*, v. 45, 11 nov. 2002.)

Agile Estimating and Planning. Descrição abrangente do planejamento baseado em histórias como é usado no XP, bem como uma justificativa para o uso de uma abordagem ágil para planejamento de projetos. Também inclui uma introdução geral, boa para questões de planejamento de projetos. (COHN, M. *Agile Estimating and Planning.* Prentice Hall, 2005.)

'Achievements and Challenges in Cocomo-based Software Resource Estimation'. Esse artigo apresenta a história dos modelos COCOMO e as influências sobre eles, além de discutir as variantes desses modelos que foram desenvolvidas. Identifica também possíveis evoluções na abordagem COCOMO. (BOEHM, B. W.; VALERIDI, R. *IEEE Software*, v. 25, n. 5, set./out. 2008.) Disponível em: <http://dx.doi.org/10.1109/MS.2008.133>.

EXERCÍCIOS

23.1 Em quais circunstâncias uma empresa poderia, justificadamente, cobrar um preço muito mais elevado do que o custo estimado de software para um sistema de software, além de uma margem de lucro razoável?

23.2 Explique por que o processo de planejamento de projeto é iterativo e por que um plano deve ser continuamente revisto durante um projeto de software.

23.3 Explique rapidamente a finalidade de cada uma das seções em um plano de projeto de software.

23.4 As estimativas de custos são inerentemente arriscadas, independentemente da técnica da estimativa usada. Sugira quatro maneiras de reduzir o risco em uma estimativa de custo.

23.5 A Tabela 23.7 define um número de tarefas, suas durações e suas dependências. Desenhe um gráfico de barras mostrando o cronograma de projeto.

23.6 A Tabela 23.7 mostra as durações de tarefas para as atividades de projeto de software. Suponha que ocorra um retrocesso grave, imprevisto e em vez de levar dez dias, a tarefa T5 leva 40 dias. Elabore novos gráficos de barras, mostrando como o projeto pode ser reorganizado.

23.7 O jogo de planejamento do XP baseia-se na noção de planejamento para implementar as histórias que representam os requisitos de sistema. Explique os potenciais problemas com essa abordagem quando o software tem altos requisitos de desempenho ou de confiança.

Tabela 23.7 Exemplos de cronograma

Tarefa	Duração (dias)	Dependências
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T2, T14
T16	10	T15

23.8 Um gerente de software é responsável pelo desenvolvimento de um sistema de software crítico de segurança, que é projetado para controlar uma máquina de radioterapia para tratar pacientes que sofrem de câncer. Esse sistema está embutido na máquina e deve ser executado em um processador de propósito especial com uma quantidade fixa de memória (256 Mbytes). A máquina comunica-se com um sistema de banco de dados de pacientes para obtenção de detalhes sobre o paciente e, após o tratamento, registra automaticamente a dose de radiação emitida e outros detalhes do tratamento no banco de dados.

O método COCOMO é usado para estimar o esforço necessário para desenvolver esse sistema e é calculada uma estimativa de 26 pessoas-mês. Ao fazer essa estimativa, todos os multiplicadores de *driver* de custo foram definidos como 1.

Explique por que essa estimativa deve ser ajustada para levar em consideração os fatores de projeto, equipe, produto e organização. Sugira quatro fatores que possam ter efeitos significativos na estimativa inicial COCOMO e proponha possíveis valores para esses fatores. Justifique a inclusão de cada fator.

23.9 Alguns projetos muito grandes de software envolvem escrever milhões de linhas de código. Explique por que os modelos de estimativa de esforço, como o COCOMO, podem não funcionar bem quando aplicados a sistemas muito grandes.

23.10 É ético para uma empresa cotar um preço baixo para um contrato de software sabendo que os requisitos são ambíguos e que eles poderão cobrar um preço elevado para mudanças posteriores solicitadas pelo cliente?

REFERÊNCIAS

BECK, K. *Extreme Programming Explained.* Reading, Mass.: Addison-Wesley, 2000.

BOEHM, B. COCOMO II Model Definition Manual. *Center for Software Engineering.* University of Southern California. 2000. Disponível em: <http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_ modelman2000.0.pdf>.

BOEHM, B.; CLARK, B.; HOROWITZ, E.; WESTLAND, C.; MADACHY, R.; SELBY, R. Cost models for future life cycle processes: COCOMO 2. *Annals of Software Engineering*, v. 1, 1995, p. 57-94.

BOEHM, B.; ROYCE, W. Ada COCOMO e the Ada Process Model. *Proc. 5th COCOMO Users' Group Meeting*, Pittsburgh: Software Engineering Institute, 1989.

BOEHM, B. W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.

BOEHM, B. W.; ABTS, C.; BROWN, A. W.; CHULANI, S.; CLARK, B. K.; HOROWITZ, E. et al. *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice Hall, 2000.

LONDEIX, B. *Cost Estimation for Software Development*. Wokingham: Addison-Wesley, 1987.

MYERS, W. Allow Plenty of Time for Large-Scale Software. *IEEE Software*, v. 6, n. 4, 1989, p. 92-99.

PUTNAM, L. H. A General Empirical Solution to the Macro Software Sizing and Estimating Problem. *IEEE Trans. on Software Engineering*, v. SE-4, n. 3, 1978, p. 345-361.

SCHWABER, K. *Agile Project Management with Scrum*. Seattle: Microsoft Press, 2004.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 **24** 25 26

Gerenciamento de qualidade

Objetivos

O objetivo deste capítulo é apresentar o gerenciamento de qualidade e a medição de software. Com a leitura deste capítulo você:

- conhecerá o processo de gerenciamento de qualidade e saberá por que o planejamento de qualidade é importante;
- saberá que a qualidade do software é afetada pelo processo usado em seu desenvolvimento;
- estará ciente da importância dos padrões no processo de gerenciamento de qualidade e saberá como os padrões são usados a fim de garantir a qualidade;
- compreenderá de que maneira revisões e inspeções são usadas como um mecanismo de garantia de qualidade de software;
- compreenderá de que maneira a medição pode ser útil na avaliação de alguns atributos de qualidade de software e as limitações atuais da medição de software.

Conteúdo

Os problemas com a qualidade de software foram inicialmente descobertos na década de 1960 com o desenvolvimento do primeiro grande sistema de software e continuaram a incomodar a engenharia de software ao longo do século XX. O software entregue era lento e pouco confiável, difícil de manter e de reusar. Em resposta à insatisfação com aquela situação, passaram a ser adotadas técnicas formais de gerenciamento de qualidade do software, as quais foram desenvolvidas a partir dos métodos usados na indústria manufatureira. Essas técnicas de gerenciamento de qualidade, em conjunto com novas tecnologias de software e melhores testes de software, conduziram a melhorias significativas no nível geral de qualidade de software.

O gerenciamento de qualidade de software para sistemas de software tem três principais preocupações:

1. No nível organizacional, o gerenciamento de qualidade está preocupado com o estabelecimento de um *framework* de processos organizacionais e padrões que levem a softwares de alta qualidade. Isso significa que a equipe de gerenciamento de qualidade deve assumir a responsabilidade de definir os processos de desenvolvimento do software que serão usados e os padrões que devem ser usados no software, bem como a documentação relacionada, incluindo os requisitos de sistema, projeto e código.
2. No nível de projeto, o gerenciamento de qualidade envolve a aplicação de processos específicos de qualidade, verificando que os processos planejados foram seguidos, e a garantia de que as saídas de projeto estejam em conformidade com os padrões aplicáveis ao projeto.

3. No nível de projeto, o gerenciamento de qualidade também está preocupado com o estabelecimento de um plano de qualidade. O plano de qualidade deve definir as metas de qualidade para o projeto e quais processos e padrões devem ser usados.

Os termos 'garantia de qualidade' e 'controle de qualidade' são amplamente usados na indústria manufatureira. A garantia de qualidade (QA, do inglês *quality assurance*) é a definição de processos e padrões que devem conduzir a produtos de alta qualidade e a introdução de processos de qualidade na fabricação. O controle de qualidade é a aplicação desses processos de qualidade visando eliminar os produtos que não atingiram o nível de qualidade exigido.

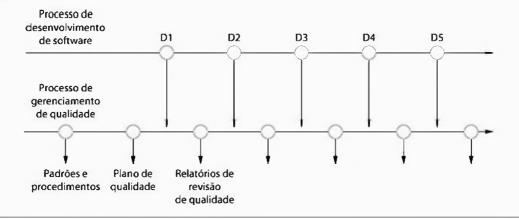
Na indústria de software, diferentes empresas e setores industriais interpretam a garantia de qualidade e controle de qualidade de maneiras diferentes. Às vezes, garantia de qualidade significa simplesmente a definição de procedimentos, processos e padrões que visam reforçar que a qualidade de software seja atingida. Em outros casos, a garantia de qualidade também inclui todo o gerenciamento de configuração, atividades de verificação e validação aplicadas após o produto ter sido entregue por uma equipe de desenvolvimento. Neste capítulo, utilizo o termo 'garantia de qualidade' para incluir a verificação e validação e os processos de verificação se os procedimentos de qualidade foram aplicados corretamente. O termo 'controle de qualidade' foi envitado, pois não é amplamente usado na indústria de software.

A equipe QA é responsável, na maioria das empresas, por gerenciar o processo de teste de *release*. Conforme discutido no Capítulo 8, isso significa que eles gerenciam os testes de software antes que este seja liberado para os clientes. Eles são responsáveis por verificar se os testes de sistema proporcionam cobertura dos requisitos e mantêm registros adequados do processo de teste. No Capítulo 8 discuti teste de *release* e, portanto, neste capítulo não discuto esse aspecto de garantia de qualidade.

O gerenciamento de qualidade fornece uma verificação independente do processo de desenvolvimento de software. O processo de gerenciamento de qualidade verifica os entregáveis de projeto para garantir que eles sejam consistentes com os padrões e objetivos organizacionais (Figura 24.1). A equipe de QA deve ser independente da equipe de desenvolvimento para que eles tenham uma visão objetiva do software. Isso lhes permite reportar sobre a qualidade de software sem sofrer influências de questões de desenvolvimento de software.

Idealmente, a equipe de gerenciamento de qualidade não deve ser associada a qualquer grupo particular de desenvolvimento, mas deve ter a responsabilidade de toda a organização para o gerenciamento de qualidade. Eles devem ser independentes e reportar para a gerência acima do nível de gerente de projeto. A razão para isso é que os gerentes de projeto precisam manter o cronograma e o orçamento de projeto. Se houver problemas, eles podem correr o risco de comprometer a qualidade do produto, em nome do cumprimento do cronograma, por exemplo. Uma equipe de gerenciamento de qualidade independente garante que os objetivos organizacionais de qualidade não se comprometam com orçamentos curtos e considerações de cronograma. No entanto, em pequenas empresas, isso é praticamente impossível. O gerenciamento de qualidade e desenvolvimento de software está inevitavelmente interligado com pessoas com responsabilidades de desenvolvimento e de qualidade.

O planejamento de qualidade é o processo de desenvolvimento de um plano de qualidade para um projeto. O plano de qualidade deve estabelecer as qualidades desejadas para o software e descrever como elas devem ser avaliadas. Portanto, define o que o software de 'alta qualidade' realmente significa para um determinado sistema. Sem essa definição, os engenheiros podem fazer suposições algumas vezes conflitantes sobre quais atributos de produto refletem as mais importantes características de qualidade. O planejamento formal de qualidade é parte integrante dos processos de desen-

Figura 24.1 Gerenciamento de qualidade e desenvolvimento de software

volvimento baseado em planos. No entanto, métodos ágeis adotam uma abordagem menos formal para o gerenciamento de qualidade.

Humphrey (1989), em seu livro clássico sobre o gerenciamento de software, sugere uma estrutura preliminar para um plano de qualidade, a qual inclui:

1. *Introdução ao produto.* Uma descrição do produto, seu mercado pretendido e as expectativas de qualidade do produto.
2. *Planos de produto.* As datas críticas de *release* e responsabilidades para o produto, junto com os planos para a distribuição e prestação de serviço do produto.
3. *Descrições de processo.* Os processos de desenvolvimento e serviço são padrões que devem ser usados para o gerenciamento e desenvolvimento de produto.
4. *Metas de qualidade.* As metas de qualidade e planos para o produto, incluindo uma identificação e uma justificativa para os atributos críticos de qualidade do produto.
5. *Riscos e gerenciamento de riscos.* Os riscos mais importantes que podem afetar a qualidade do produto e as ações que devem ser tomadas ao lidar com eles.

Os planos de qualidade, os quais são desenvolvidos como parte do processo geral de planejamento de projeto, diferem em detalhes, dependendo do tamanho e do tipo de sistema que está sendo desenvolvido. No entanto, ao escrever planos de qualidade, você deve tentar mantê-los o mais breve possível. Se o documento for muito extenso, as pessoas não o lerão, e isso reduz o propósito de se produzir o plano de qualidade.

Algumas pessoas pensam que a qualidade de software pode ser alcançada por meio de processos prescritivos, baseados em padrões organizacionais e procedimentos de qualidade associados que verificam que esses padrões serão seguidos pela equipe de desenvolvimento de software. Seu argumento é que os padrões incorporam as boas práticas de engenharia de software e que segui-las levará a produtos de alta qualidade. Na prática, contudo, existe muito mais no gerenciamento de qualidade do que apenas padrões e a burocracia associada para garantir que sejam seguidos.

Padrões e processos são importantes, mas os gerentes de qualidade também devem ter como objetivo desenvolver uma 'cultura de qualidade' em que todos os responsáveis pelo desenvolvimento de software estejam comprometidos em alcançar um alto nível de qualidade de produto. Eles devem incentivar as equipes a assumirem responsabilidade pela qualidade de seu trabalho e a desenvolverem novas abordagens para a melhoria de qualidade. Apesar de os padrões e procedimentos serem a base do gerenciamento de qualidade, os bons gerentes de qualidade reconhecem haver aspectos intangíveis para a qualidade de software (elegância, legibilidade etc.) que não podem ser incorporados em padrões. Eles devem apoiar as pessoas que estão interessadas em aspectos intangíveis de qualidade e incentivar o comportamento profissional de todos os membros de equipe.

O gerenciamento formal de qualidade é particularmente importante para as equipes que estão desenvolvendo sistemas de longa vida e grande porte que levam vários anos para se desenvolver. A documentação de qualidade é um registro do que foi feito por cada subgrupo no projeto. Isso ajuda as pessoas a verificarem se tarefas importantes não foram esquecidas ou se um grupo não fez suposições incorretas sobre o trabalho de outros grupos. A documentação de qualidade também é um meio de comunicação durante a vida útil de um sistema. Ela permite que os grupos responsáveis pela evolução de sistema possam rastrear os testes e as verificações implementadas pela equipe de desenvolvimento.

Para sistemas menores, o gerenciamento de qualidade também é importante, mas pode adotar uma abordagem mais informal. Não é necessária muita papelada porque uma equipe pequena de desenvolvimento também pode comunicar-se informalmente. O ponto mais importante do gerenciamento de qualidade para o desenvolvimento de pequenos sistemas é estabelecer uma cultura de qualidade e assegurar que todos os membros de equipe tenham uma abordagem positiva da qualidade de software.

24.1 Qualidade de software

Os fundamentos do gerenciamento de qualidade foram estabelecidos pela indústria manufatureira em um esforço para melhorar a qualidade dos produtos em produção. Como parte disso, eles desenvolveram uma definição de 'qualidade', baseada na conformidade com uma especificação detalhada (CROSBY, 1979) e na noção de tolerâncias. A suposição subjacente foi a de que os produtos poderiam ser completamente especificados e poderiam ser estabelecidos procedimentos para avaliar um produto fabricado de acordo com suas especificações. Naturalmen-

te, os produtos nunca cumprirão todas as especificações, então se admitiu alguma tolerância. Se o produto estava 'quase certo', ele era classificado como aceitável.

A qualidade de software não é diretamente comparável à qualidade na manufatura. A ideia de tolerâncias não é aplicável aos sistemas digitais e, pelas razões apresentadas a seguir, pode ser impossível concluir objetivamente se um sistema de software cumpre ou não suas especificações:

1. Conforme discutido no Capítulo 4, no qual abordo a engenharia de requisitos, é difícil escrever especificações de software completas e precisas. Os clientes e desenvolvedores de software podem interpretar os requisitos de maneiras diferentes e pode ser impossível chegar a um acordo sobre se o software cumpre ou não suas especificações.
2. Geralmente, as especificações integram requisitos de várias classes de *stakeholders*. Esses requisitos são, inevitavelmente, um compromisso e podem não incluir os requisitos de todos os grupos de *stakeholders*. Os *stakeholders* excluídos podem perceber o sistema como um sistema de baixa qualidade, mesmo que este implemente os requisitos acordados.
3. É impossível medir determinadas características de qualidade diretamente (por exemplo, manutenibilidade), assim, elas não podem ser especificadas de forma não ambígua. As dificuldades de medição são discutidas na Seção 24.4.

Devido a esses problemas, a avaliação da qualidade de software é um processo subjetivo, em que a equipe de gerenciamento de qualidade precisa usar seu julgamento para decidir se foi alcançado um nível aceitável de qualidade. Ela precisa considerar se o software é adequado para sua finalidade ou não. Trata-se de responder a perguntas sobre as características do sistema. Por exemplo:

1. Durante o processo de desenvolvimento os padrões de programação e documentação foram seguidos?
2. O software foi devidamente testado?
3. O software é suficientemente confiável para ser colocado em uso?
4. O desempenho do software é aceitável para uso normal?
5. O software é útil?
6. O software é bem estruturado e compreensível?

Existe uma suposição geral no gerenciamento de qualidade de software de que os testes de sistema serão baseados em seus requisitos. A decisão sobre se o software oferece ou não a funcionalidade necessária deve basear-se nos resultados desses testes. Por isso, a equipe de QA deve revisar os testes que foram desenvolvidos e analisar os registros de testes para verificar se os testes foram devidamente realizados. Em algumas organizações, a equipe de gerenciamento de qualidade é responsável pelos testes de sistema, mas, às vezes, essa responsabilidade é dada para um grupo separado de testes.

A qualidade subjetiva de um sistema de software baseia-se em grande parte em suas características não funcionais. Isso reflete a experiência prática do usuário — se a funcionalidade do software não é a esperada, os usuários frequentemente apenas contornam esse problema e encontram outras maneiras de fazer o que querem. No entanto, se o software for muito lento ou não confiável, será praticamente impossível aos usuários atingirem seus objetivos.

Portanto, entendemos que a qualidade de software não implica apenas se a funcionalidade de software foi corretamente implementada, mas também depende dos atributos não funcionais de sistema. Boehm et al. (1978) sugeriram que havia quinze atributos importantes de qualidade de software, como mostrado na Tabela 24.1. Esses atributos estão relacionados com a confiança, a usabilidade, a eficiência e a manutenibilidade de software. Como já discutido no Capítulo 11, geralmente os atributos de confiança são os atributos de qualidade mais importantes de um sistema. No entanto, o desempenho do software também é muito importante. Os usuários rejeitarão o software que for muito lento.

É impossível que algum sistema seja otimizado em todos esses atributos — por exemplo, melhorar a robustez pode levar à perda de desempenho. O plano de qualidade, portanto, deve definir os atributos de qualidade mais importantes para o software que está sendo desenvolvido. Pode ser que a eficiência seja crítica e outros fatores tenham de ser sacrificados para obtenção disso. Se no plano de qualidade você estabeleceu que um atributo é crítico, os engenheiros que trabalham no desenvolvimento podem cooperar para esse atributo. O plano deve incluir também uma definição de processo de avaliação de qualidade. Isso deve ser uma maneira de avaliar se alguma qualidade, como manutenibilidade ou robustez, está presente no produto.

Um pressuposto do gerenciamento de qualidade de software é que a qualidade do software é diretamente relacionada à qualidade do processo de desenvolvimento de software. Isso vem novamente da fabricação de sistemas, em que a qualidade de produto é intimamente relacionada ao processo de produção.

Tabela 24.1 Atributos de qualidade de software

Segurança	Compreensibilidade	Portabilidade
Proteção	Testabilidade	Usabilidade
Confiabilidade	Adaptabilidade	Reusabilidade
Resiliência	Modularidade	Eficiência
Robustez	Complexidade	Capacidade de aprendizado

Um processo de fabricação envolve configurar e operar as máquinas envolvidas no processo. Uma vez que as máquinas estão funcionando corretamente, a qualidade de produto segue normalmente. Você mede a qualidade do produto e altera o processo até atingir o nível de qualidade requerida. A Figura 24.2 ilustra essa abordagem baseada em processos para atingir a qualidade de produto.

Existe uma clara ligação entre a qualidade de processo e de produto na manufatura porque o processo é relativamente fácil de ser padronizado e monitorado. Uma vez que sistemas de manufatura sejam calibrados, eles podem ser executados várias vezes para produzir produtos de alta qualidade. No entanto, o software não é manufaturado — ele é criado. No desenvolvimento de software a relação entre a qualidade de processo e de produto é mais complexa. O desenvolvimento de software é um processo criativo, em vez de mecânico, portanto, a influência de competências e experiências individuais é significativa. Os fatores externos, como a novidade de uma aplicação ou pressão comercial para um *release* anterior do produto, também afetam a qualidade de produto, independentemente do processo usado.

Não há dúvida de que o processo de desenvolvimento usado tenha uma influência significativa sobre a qualidade de software e que bons processos são mais suscetíveis de conduzir o software de boa qualidade. O gerenciamento e a melhoria de qualidade de processo podem gerar softwares com menos defeitos. Contudo, é difícil avaliar os atributos de qualidade de software, como manutenibilidade, sem usar o software por um longo período. Portanto, é difícil dizer como as características de processo influenciam esses atributos. Além disso, por causa do papel do projeto e da criatividade no processo de software, a padronização de processos pode, por vezes, sufocar a criatividade, o que pode gerar softwares com menos qualidade.

24.2 Padrões de software

Os padrões de software desempenham um papel muito importante no gerenciamento de qualidade de software. Como já discutido, uma parte importante da garantia de qualidade é a definição ou seleção de padrões que devem ser aplicados no processo de desenvolvimento de software ou produtos de software. Como parte desse processo de QA, devem ser escolhidas ferramentas e métodos para suportar o uso desses padrões. Uma vez que os padrões foram selecionados para uso, processos específicos de projeto devem ser definidos para monitorar o uso dos padrões e verificar se eles foram seguidos.

Figura 24.2 Qualidade baseada em processos

Os padrões de software são importantes por três razões:

1. Capturam sabedoria, que é valiosa para a organização. Eles são baseados em conhecimentos sobre a prática do que é melhor ou mais adequado para a empresa. Muitas vezes, esse conhecimento é adquirido após bastante tentativa e erro. Inseri-lo em um padrão ajuda a empresa a reusar essa experiência e evitar erros anteriormente cometidos.
2. Fornecem um *framework* para a definição do significado de 'qualidade' em uma determinada organização. Como já discutido, a qualidade de software é subjetiva e ao usar padrões você estabelece uma base para decidir se o nível de qualidade exigido foi atingido. Naturalmente, isso depende do estabelecimento de padrões que reflitam as expectativas do usuário com relação a confiança, usabilidade e desempenho do software.
3. Ajudam a dar continuidade ao trabalho realizado por uma pessoa, quando retomado e continuado por outra. Os padrões asseguram que todos os engenheiros dentro de uma organização adotem as mesmas práticas. Consequentemente, o esforço de aprendizagem requerido ao iniciar um novo trabalho é reduzido.

Existem dois tipos de padrões de engenharia de software que podem ser definidos e usados no gerenciamento de qualidade de software:

1. *Padrões de produto.* Aplicam-se ao produto de software que está sendo desenvolvido. Eles incluem padrões de documentos — como a estrutura dos documentos de requisitos; padrões de documentação — como um cabeçalho de comentário padrão para uma definição de classe de objeto; e padrões de codificação — os quais definem como uma linguagem de programação deve ser usada.
2. *Padrões de processo.* Definem os processos que devem ser seguidos durante o desenvolvimento de software. Eles devem encapsular as boas práticas de desenvolvimento. Os padrões de processo podem incluir definições de especificação, projeto e processos de validação, ferramentas de suporte do processo e uma descrição dos documentos que devem ser escritos durante esses processos.

Os padrões devem entregar valor, sob a forma de maior qualidade de produto. Não há razão para definir padrões que sejam caros em termos de tempo e de esforço para serem aplicados e que gerem poucas melhorias na qualidade. Os padrões de produto precisam ser projetados para poderem ser aplicados e verificados de forma efetiva e os padrões de processos devem incluir uma definição de processos que verifique se os padrões de produto foram seguidos.

Geralmente, o desenvolvimento de padrões internacionais de engenharia de software é um processo prolongado, em que os interessados no padrão se encontram, produzem material para discussões e, finalmente, chegam a um acordo sobre o padrão. Organismos nacionais e internacionais, como DoD dos Estados Unidos, ANSI, BSI, OTAN e IEEE apoiam a produção de padrões. Tratam-se de padrões gerais que podem ser aplicados em uma gama de projetos. Organismos como a OTAN e outras organizações de defesa podem exigir que seus próprios padrões sejam usados em seus contratos de desenvolvimento com empresas de software.

Foram desenvolvidos padrões nacionais e internacionais, discutindo a terminologia de engenharia de software e linguagens de programação como Java e C++, notações, como gráficos, símbolos, procedimentos para derivar e escrever requisitos de software, procedimentos de garantia de qualidade e processos de verificação e validação de software (IEEE, 2003). Os padrões mais especializados, como IEC 61508 (IEC, 1998), foram desenvolvidos para sistemas críticos de segurança e proteção.

As equipes de gerenciamento de qualidade que estão desenvolvendo padrões para uma empresa devem, em geral, basear esses padrões em padrões nacionais e internacionais. Ao usar padrões internacionais como ponto de partida, a equipe de garantia de qualidade deve elaborar um 'manual' de padrões. Este deve definir os padrões necessários para sua organização. Exemplos de padrões que poderiam ser incluídos nesse manual são mostrados na Tabela 24.2.

Algumas vezes, os engenheiros de software consideram os padrões demasiadamente prescritivos e não realmente relevantes para a atividade técnica de desenvolvimento de software. Isso é mais provável quando os padrões de projeto exigem documentação tediosa e registros de trabalho. Embora geralmente concordem sobre a necessidade geral de padrões, muitas vezes os engenheiros encontram boas razões pelas quais padrões não se adequam a seus projetos. Portanto, para minimizar o descontentamento e encorajar o uso de padrões, os gerentes de qualidade que definem os padrões devem seguir os seguintes passos:

1. *Envolver os engenheiros de software na seleção de padrões de produto.* Se os desenvolvedores entenderem por que os padrões foram selecionados, eles estarão mais propensos a se comprometerem com esses padrões. Idealmente, além de definir o padrão a ser seguido, o documento de padrões também deve incluir comentários explicando por que essas decisões de padronização foram tomadas.

Tabela 24.2 Padrões de produto e de processo

Padrões de produto	**Padrões de processo**
Formulário de revisão de projeto	Condução de revisão de projeto
Estrutura de documento de requisitos	Apresentação do novo código para a construção de sistema
Formato de cabeçalho de método	Processo de versão e *release*
Estilo de programação Java	Processo de aprovação de plano de projeto
Formato de plano de projeto	Processo de controle de mudança
Formulário de solicitação de mudança	Processo de registro de teste

2. *Revisar e modificar regularmente os padrões para refletir mudanças nas tecnologias.* O desenvolvimento de padrões é um processo caro e os padrões tendem a ser consagrados em um manual de padrões de empresa. Por causa dos custos e das discussões necessárias, muitas vezes existe relutância em alterá-los. Um manual de padrões é essencial, mas ele deve evoluir para refletir mudanças nas circunstâncias e tecnologias.
3. *Fornecer ferramentas de software para oferecer suporte aos padrões.* Muitas vezes, os desenvolvedores percebem os padrões como um estorvo, em casos nos quais a conformidade a esses padrões envolve tediosos trabalhos manuais que poderiam ser realizados por uma ferramenta de software. Se o suporte a ferramentas está disponível, é necessário muito pouco esforço para seguir os padrões de desenvolvimento de software. Por exemplo, padrões de documentos podem ser implementados usando-se estilos de processador de texto.

Os diferentes tipos de software precisam de diferentes processos de desenvolvimento, portanto, os padrões preciam ser adaptáveis. Não faz sentido prescrever um modo particular de trabalho se ele é inadequado a um projeto ou a uma equipe de projeto. Cada gerente de projeto deve ter autoridade para modificar padrões de processo de acordo com as circunstâncias individuais. No entanto, quando são feitas alterações, é importante assegurar que essas alterações não ocasionem a perda da qualidade de produto. Isso afetaria o relacionamento da empresa com seus clientes e provavelmente ocasionaria custos maiores para o projeto.

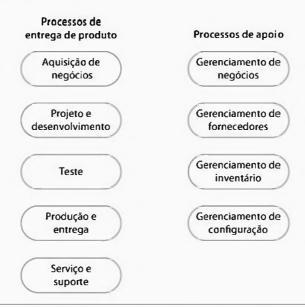
O gerente de projeto e o gerente de qualidade podem evitar os problemas com padrões não apropriados por meio de um planejamento da qualidade cuidadoso logo no início do projeto. Eles devem decidir quais padrões organizacionais devem ser usados sem alterações, quais devem ser modificados e quais devem ser ignorados. Novos padrões podem ser necessários em resposta aos requisitos de clientes e de projeto. Por exemplo, padrões para especificações formais podem ser necessários caso estas não tenham sido usadas em projetos anteriores.

24.2.1 O *framework* de normas ISO 9001

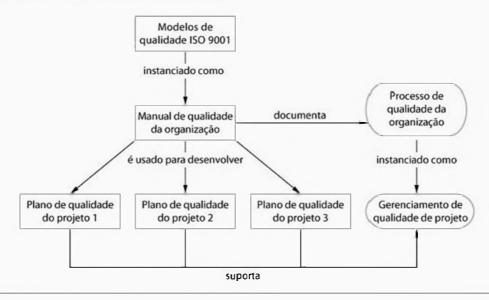
Existe uma série de normas que podem ser usadas no desenvolvimento de sistemas de gerenciamento de qualidade em todos os setores, chamada ISO 9000. As normas ISO 9000 podem ser aplicadas a uma variedade de organizações, desde a produção até a indústria de serviços. A ISO 9001, a mais geral desses padrões, aplica-se a organizações que projetam, desenvolvem e mantêm produtos, incluindo software. Originalmente, a norma ISO 9001 foi desenvolvida em 1987, com sua mais recente revisão em 2008.

A norma ISO 9001 não é propriamente um padrão para o desenvolvimento de software, mas é um *framework* para o desenvolvimento de padrões de software. Ela define os princípios gerais da qualidade, descreve os processos gerais de qualidade e estabelece os padrões organizacionais e os procedimentos que devem ser definidos. Estes devem ser documentados em um manual de qualidade da organização.

A grande revisão da norma ISO 9001 em 2000 reorientou os processos em torno de nove padrões principais (Figura 24.3). Se uma organização deve adequar-se à ISO 9001, deve documentar como seus processos se referem a esses processos essenciais. Essa organização também deve definir e manter registros que demonstrem que os processos organizacionais definidos foram seguidos. O manual de qualidade da empresa deve descrever os processos relevantes e os dados de processo que devem ser coletados e mantidos.

Figura 24.3 Os processos essenciais da ISO 9001

As normas ISO 9001 não definem ou prescrevem os processos de qualidade específicos que devem ser usados em uma empresa. Para estar adequada à ISO 9001, uma empresa deve ter definidos os tipos de processos mostrados na Figura 24.3, além de ter procedimentos que demonstrem que seus processos de qualidade estão sendo seguidos. Isso permite flexibilidade entre setores industriais e tamanhos de empresa. Podem ser definidos padrões de qualidade apropriados para o tipo de software que está sendo desenvolvido. As pequenas empresas podem ter processos não burocráticos e ainda estarem em conformidade com a ISO 9001. No entanto, essa flexibilidade significa que você não pode fazer suposições sobre as semelhanças ou diferenças entre os processos em diferentes conformidades à ISO 9001. Algumas empresas podem ter processos de qualidade muito rígidos que mantenham registros detalhados, enquanto outras podem ser menos formais, com documentação adicional mínima.

Os relacionamentos entre a ISO 9001, os manuais de qualidade organizacional e os planos individuais de qualidade de projeto são mostrados na Figura 24.4. Esse diagrama foi obtido a partir de um modelo dado por Ince (1994), que explica como a norma geral ISO 9001 pode ser usada como uma base para processos de gerenciamento de qualidade de software. Bamford e Dielbler (2003) explicaram como a norma posterior ISO 9001:2000, pode ser aplicada em empresas de software.

Figura 24.4 ISO 9001 e gerenciamento de qualidade

Alguns clientes de software exigem que seus fornecedores sejam certificados pela ISO 9001. Dessa forma, eles podem ficar confiantes de que a empresa de desenvolvimento de software tem um sistema de gerenciamento de qualidade aprovado. Autoridades independentes de certificação examinam os processos de gerenciamento de qualidade, processam a documentação e decidem se esses processos cobrem todas as áreas especificadas na ISO 9001. Caso isso ocorra, elas certificam que os processos de qualidade da empresa, como definidos no manual de qualidade, estão em conformidade com a norma ISO 9001.

Algumas pessoas pensam que a certificação ISO 9001 significa que a qualidade do software produzido por empresas certificadas será melhor do que o software produzido em empresas não certificadas. Isso não é necessariamente verdade. A norma ISO 9001 assegura que a organização dispõe de procedimentos de gerenciamento de qualidade e segue tais procedimentos. Não há uma garantia de que empresas certificadas com a ISO 9001 usam as melhores práticas de desenvolvimento de software ou que seus processos gerarão software com alta qualidade.

Por exemplo, uma empresa poderia definir padrões de cobertura de teste especificando que todos os métodos em objetos devem ser chamados ao menos uma vez. Infelizmente, essa norma pode ser atendida por testes de software incompleto, que não executam testes com diferentes parâmetros de método. Desde que sejam seguidos os procedimentos de testes e mantidos os registos dos testes realizados, a empresa poderia ser certificada com a ISO 9001. A certificação ISO 9001 define como qualidade a conformidade com os padrões e não leva em consideração a qualidade vivida pelos usuários do software.

Os métodos ágeis, que evitam a documentação e se concentram no código que está sendo desenvolvido, têm pouco em comum com os processos de qualidade formais discutidos na ISO 9001. Já existe algum trabalho no sentido de reconciliar essas abordagens (STALHANE e HANSSEN, 2008), mas a comunidade de desenvolvimento ágil é fundamentalmente contra ao que eles veem como *overhead* burocrático de conformidade a normas. Por essa razão, empresas que usam métodos ágeis de desenvolvimento raramente estão preocupadas com a certificação ISO 9001.

24.3 Revisões e inspeções

Revisões e inspeções são atividades de controle de qualidade que verificam a qualidade dos entregáveis de projeto. Isso envolve examinar o software, sua documentação e os registros do processo para descobrir erros e omissões e verificar se os padrões de qualidade foram seguidos. Conforme discutido nos capítulos 8 e 15, revisões e inspeções são usadas junto com teste de programa, como parte do processo geral de validação e verificação de software.

Durante uma revisão, um grupo de pessoas examina o software e a documentação associada à procura de possíveis problemas e não conformidade com padrões. A equipe de revisão informada sobre o nível de qualidade de um sistema ou entregável de projeto toma decisões. Os gerentes de projeto podem usar essas avaliações para tomar decisões de planejamento e alocar recursos para o processo de desenvolvimento.

As avaliações de qualidade são baseadas em documentos que foram produzidos durante o processo de desenvolvimento de software. Assim como as especificações de software, projetos ou códigos, modelos de processos, planos de testes, procedimentos de gerenciamento de configuração, padrões de processo e manuais de usuário também podem ser revistos. A revisão deve verificar a consistência e a completude dos documentos ou código em revisão e certificar-se de que os padrões de qualidade foram seguidos.

No entanto, as revisões não servem apenas para verificar a conformidade com os padrões. Elas também são usadas para ajudar a descobrir problemas e omissões no software ou na documentação de projeto. As conclusões das revisões devem ser formalmente registradas como parte do processo de gerenciamento de qualidade. Se os problemas forem descobertos, os comentários dos revisores devem ser enviados ao autor do software ou a quem estiver responsável por corrigir os erros ou as omissões.

O objetivo das revisões e inspeções é melhorar a qualidade de software e não avaliar o desempenho das pessoas na equipe de desenvolvimento. A revisão é um processo público de detecção de erros, em comparação com processos de testes de componente mais privados. Inevitavelmente, erros que são cometidos por indivíduos são revelados a toda a equipe de programação. Para garantir que todos os desenvolvedores estejam engajados construtivamente com o processo de revisão, os gerentes de projeto precisam estar sensíveis às preocupações individuais. Eles precisam desenvolver uma cultura de trabalho que forneça suporte sem procurar os culpados pelos erros descobertos.

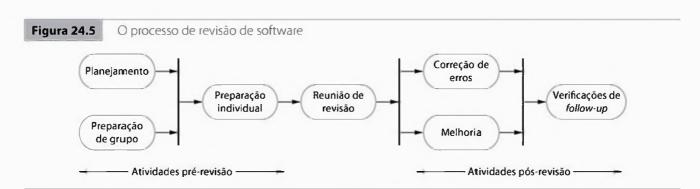
Ainda que uma revisão de qualidade forneça informações sobre o software que está sendo desenvolvido para gerenciamento, não é igual a uma revisão de progresso de gerenciamento. Conforme discutido no Capítulo 23, as revisões de progresso comparam o andamento real de um projeto de software com o progresso planejado. Sua principal preocupação é o projeto entregar o software útil no prazo e dentro do orçamento. As revisões de progresso levam em conta os fatores externos e as alterações de circunstâncias que podem significar que o software em desenvolvimento não é mais necessário ou precisa ser radicalmente alterado. Projetos que desenvolveram softwares de alta qualidade podem ser cancelados por mudanças nos negócios ou em seu ambiente operacional.

24.3.1 O processo de revisão

Embora existam muitas variações nos detalhes das revisões, o processo de revisão (Figura 24.5) normalmente é estruturado em três fases:

1. *Atividades pré-revisão*. As atividades preparatórias são essenciais para a eficácia da revisão. Em geral, as atividades de pré-revisão estão relacionadas com o planejamento e a preparação da revisão. O planejamento de revisão envolve a definição de uma equipe de revisão, a organização de um tempo e de um lugar para sua ocorrência e a distribuição de documentos a serem revistos. Durante a preparação da revisão, a equipe pode reunir-se para obter uma visão geral do software a ser revisto. Os membros da equipe de revisão individual precisam ler e entender o software ou os documentos e padrões relevantes. Eles trabalham independentemente para encontrar erros, omissões e desvios de padrões. Os revisores podem fornecer comentários escritos sobre o software, caso eles não possam participar da reunião de revisão.
2. *Reunião de revisão*. Durante a reunião de revisão, o autor do documento ou do programa a ser revisto deve 'caminhar' pelo documento com a equipe de revisão. A revisão em si deve ser relativamente curta — duas horas, no máximo. Um membro da equipe deve presidir a revisão e outro deve registrar formalmente todas as decisões e ações a serem tomadas. Durante a revisão, o presidente é responsável por garantir que todos os comentários escritos sejam considerados. O presidente da revisão deve escrever um registro dos comentários e ações acordadas durante a revisão.
3. *Atividades pós-revisão*. Após a reunião de revisão, as questões e os problemas levantados devem ser abordados. Esse processo pode envolver a correção de *bugs* de software, a refatoração do software para que ele esteja em conformidade com os padrões de qualidade, ou a necessidade de uma nova redação dos documentos. Em alguns casos, os problemas descobertos em uma revisão de qualidade são tais que uma revisão de gerenciamento também é necessária para decidir se mais recursos devem ser disponibilizados para corrigi-los. Após as alterações terem sido feitas, o presidente da revisão pode verificar se todos os comentários de revisão foram levados em consideração. Às vezes, uma nova revisão será necessária para verificar se as alterações efetuadas cobriram todos os comentários da revisão anterior.

Geralmente, as equipes de revisão devem ter um núcleo com três a quatro pessoas, selecionadas como principais revisores. Um membro deve ser um projetista sênior, o qual deve assumir a responsabilidade pela tomada de decisões técnicas significativas. Os revisores principais podem convidar outros membros de projeto, como os projetistas de subsistemas relacionados, para contribuir com a revisão. Eles não podem estar envolvidos na revisão completa do documento, mas devem concentrar-se nas seções que afetam seu trabalho. Como alternativa, a equipe de revisão pode circular o documento e solicitar comentários por escrito de vários membros de projeto. O gerente de projeto não precisa estar envolvido na revisão, a menos que se esperem problemas que exijam alterações no plano de projeto.

Figura 24.5 O processo de revisão de software

O processo de revisão exemplificado depende de todos os membros de uma equipe de desenvolvimento estarem reunidos e disponíveis para uma reunião. No entanto, atualmente, as equipes de projeto costumam ser distribuídas geograficamente, por vezes em países ou continentes diferentes, e, em virtude disso, pode ser inviável a reunião de membros de equipe. Em tais situações, ferramentas de edição de documento podem ser usadas para apoiar o processo de revisão. Os membros de equipe podem usá-las para fazer anotações no documento ou comentários no código-fonte de software. Esses comentários são visíveis a outros membros de equipe, os quais podem, em seguida, aprová-los ou rejeitá-los. Uma discussão por telefone pode ser necessária quando for preciso resolver divergências entre os revisores.

Em desenvolvimento ágil, o processo de revisão de software normalmente é informal. No Scrum, por exemplo, ocorre uma reunião de revisão após a conclusão da cada iteração do software (uma revisão de *sprint*), em que os problemas e as questões de qualidade podem ser discutidos. Em Extreme Programming, conforme discuto na próxima seção, a programação em pares garante que o código esteja sendo examinado e revisto constantemente por outro membro de equipe. As questões de qualidade geral também são consideradas nas reuniões diárias de equipe, mas XP baseia-se em indivíduos que tomam a iniciativa para melhorar e refatorar o código. Abordagens ágeis não costumam ser dirigidas a padrões; desse modo, as questões de conformidade com padrões geralmente não são consideradas.

A falta de procedimentos formais de qualidade em métodos ágeis significa que pode haver problemas ao se usarem as abordagens ágeis em empresas que desenvolveram procedimentos detalhados de gerenciamento de qualidade. Revisões de qualidade podem diminuir o ritmo de desenvolvimento de software e elas são melhor usadas em um processo de desenvolvimento dirigido a planos, no qual as revisões podem ser planejadas e um outro trabalho, programado em paralelo. Isso é impraticável em abordagens ágeis centradas unicamente no desenvolvimento de código.

24.3.2 Inspeções de programa

As inspeções de programa são 'revisões em pares' em que os membros da equipe colaboram para encontrar *bugs* no programa que está sendo desenvolvido. Conforme discutido no Capítulo 8, as inspeções podem fazer parte dos processos de verificação e validação de software. Elas complementam os testes, pois não exigem que o programa seja executado. Isso significa que podem ser verificadas versões incompletas do sistema e que representações, tais como modelos UML, podem ser checadas. Gilb e Graham (1993) sugerem que uma das maneiras mais efetivas de se usar as inspeções é revisar os casos de teste para um sistema. As inspeções podem descobrir problemas com testes e, assim, melhorar a eficácia desses testes em detectar *bugs* no programa.

As inspeções de programa envolvem membros de equipe de diferentes origens fazendo uma revisão cuidadosa, linha por linha de código-fonte de programa. Eles procuram defeitos e problemas e os descrevem em uma reunião de inspeção. Os defeitos podem ser erros lógicos, anomalias no código que podem indicar uma condição errada ou recursos que foram omitidos do código. A equipe de revisão examina em detalhes os modelos de projeto ou o código de programa e destaca anomalias e problemas para que sejam reparados.

Durante uma inspeção, frequentemente se usa um *checklist* de erros comuns de programação para ajudar na busca por *bugs*. Esse *checklist* pode basear-se em exemplos de livros ou no conhecimento de defeitos comuns em um domínio de aplicação específico. Diferentes *checklists* são usados para diferentes linguagens de programação, pois cada linguagem tem seus próprios erros característicos. Humphrey (1989), em uma discussão abrangente de inspeções, dá uma série de exemplos de *checklists* de inspeção.

Algumas possíveis verificações, as quais podem ser feitas durante o processo de inspeção, são mostradas na Tabela 24.3. Gilb e Graham (1993) enfatizam que cada organização deve desenvolver seu próprio *checklist* de inspeção com base em práticas e padrões locais. Esses *checklists* devem ser atualizados regularmente, à medida que novos tipos de defeitos são encontrados. Os itens no *checklist* variam de acordo com a linguagem de programação, em virtude dos diferentes níveis de verificação possíveis em tempo de compilação. Por exemplo, um compilador Java verifica se as funções têm o número correto de parâmetros; um compilador C, por sua vez, não verifica.

A maioria das empresas que introduziu inspeções descobriu que estas são muito eficazes em encontrar *bugs*. Fagan (1986) informou que mais de 60% dos erros em um programa podem ser detectados por meio de inspeções informais de programa. Mills et al. (1987) sugerem que uma abordagem mais formal para inspeção, com base em argumentos de correção, pode detectar mais de 90% dos erros em um programa. McConnell (2004) compara testes de unidade, em que a taxa de detecção de defeitos é de cerca de 25%, com inspeções, em que a taxa de detecção de defeitos era de 60%. Ele também descreve uma série de estudos de caso, incluindo um exemplo em que a introdução de revisões em pares levou a um aumento de 14% na produtividade e diminuiu em 90% os defeitos de programa.

Tabela 24.3 Um *checklist* de inspeção

Classe de defeito	Verificação de inspeção
Defeitos de dados	• Todas as variáveis de programa são iniciadas antes que seus valores sejam usados? • Todas as constantes foram nomeadas? • O limite superior de vetores deve ser igual ao tamanho do vetor ou ao tamanho −1? • Se as *strings* de caracteres são usadas, um delimitador é explicitamente atribuído? • Existe alguma possibilidade de *overflow* de *buffer*?
Defeitos de controle	• Para cada instrução condicional, a condição está correta? • É certo que cada *loop* vai terminar? • As declarações compostas estão posicionadas corretamente entre colchetes? • Em declarações *case*, todos os *cases* possíveis são considerados? • Se um *break* é requerido após cada *case* em declarações *case*, este foi incluído?
Defeitos de entrada/saída	• Todas as variáveis de entrada são usadas? • Todas as variáveis de saída receberam um valor antes de serem emitidas? • Entradas inesperadas podem causar corrupção de dados?
Defeitos de interface	• Todas as chamadas de funções e métodos têm o número correto de parâmetros? • Os parâmetros formais e reais correspondem? • Os parâmetros estão na ordem correta? • Se os componentes acessam memória compartilhada, eles têm o mesmo modelo de estrutura de memória compartilhada?
Defeitos de gerenciamento de armazenamento	• Se uma estrutura ligada é modificada, todas as ligações foram corretamente reatribuídas? • Se o armazenamento dinâmico é usado, o espaço foi alocado corretamente? • O espaço é explicitamente desalocado após não ser mais necessário?
Defeitos de gerenciamento de exceção	• Foram levadas em consideração todas as condições possíveis de erro?

Apesar da efetividade reconhecida, muitas empresas de desenvolvimento de software são relutantes em usar inspeções ou revisões em pares. Engenheiros de software com experiência em testes de programa, por vezes, não estão dispostos a aceitar que inspeções possam ser mais eficazes que os testes na detecção de defeitos. Gerentes podem ser suspeitos, pois inspeções exigem custos adicionais durante o projeto e desenvolvimento. Eles podem não querer correr o risco de não haver redução nos custos de testes de programa.

Os processos ágeis raramente usam processos formais de inspeção ou de revisão em pares. Em vez disso, eles contam com os membros de equipe que colaboram para verificar o código uns dos outros e as diretrizes informais, como 'verifique antes do *check-in*', que sugerem que os programadores devem verificar seu próprio código. Os adeptos de Extreme Programming argumentam que a programação em pares é um substituto eficaz para inspeção, pois é um processo contínuo de inspeção. Duas pessoas olham cada linha de código e verificam antes de este ser aceito.

A programação em pares leva ao profundo conhecimento de um programa, pois ambos os programadores precisam compreender seu funcionamento em detalhes para continuar o desenvolvimento. Esse conhecimento, às vezes, é difícil de alcançar em outros processos de inspeção, assim, a programação em pares pode encontrar *bugs* que, às vezes, não são descobertos em inspeções formais. No entanto, a programação em pares também pode levar a desentendimentos mútuos de requisitos, em que ambos os membros do par cometem o mesmo erro. Além disso, os pares podem estar relutantes em procurar por erros, por não quererem diminuir o ritmo do andamento do projeto. As pessoas envolvidas não podem ser tão objetivas como uma equipe de inspeção externa e sua capacidade de detectar defeitos pode ser comprometida por seu estreito relacionamento de trabalho.

24.4 Medições e métricas de software

A medição de software preocupa-se com a derivação de um valor numérico ou o perfil para um atributo de um componente de software, sistema ou processo. Comparando esses valores entre si e com os padrões que se

aplicam a toda a organização, você pode ser capaz de tirar conclusões sobre a qualidade do software ou avaliar a eficácia dos métodos, das ferramentas e dos processos de software.

Por exemplo, digamos que uma organização tem a intenção de introduzir uma nova ferramenta de teste de software. Antes de introduzir a ferramenta, em um determinado momento você registra o número de defeitos de software descobertos. Essa é uma *baseline* de avaliação da eficácia da ferramenta. Depois de algum tempo usando a ferramenta, esse processo é repetido. Se mais defeitos forem encontrados durante o mesmo período, depois que a ferramenta foi introduzida, você pode decidir se ela fornece suporte útil para o processo de validação de software.

O objetivo a longo prazo de medição de software é usá-la no lugar de revisões para fazer julgamentos sobre a qualidade de software. Usando a medição de software, um sistema poderia, idealmente, ser avaliado usando uma variedade de métricas e, a partir dessa medição, deduzir um valor para a qualidade do sistema. Se o software atingir o limiar de qualidade requerido, então ele poderia ser aprovado sem revisão. Quando apropriado, as ferramentas de medição também podem realçar áreas do software que poderiam ser melhoradas. No entanto, estamos ainda longe dessa situação ideal e não há sinal de que as avaliações automatizadas de qualidade venham a se tornar realidade em um futuro próximo.

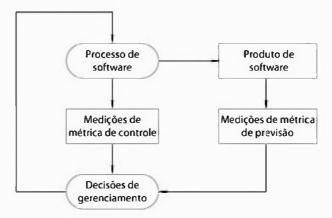
Uma métrica de software é uma característica de um sistema de software, documentação de sistema ou processo de desenvolvimento que pode ser objetivamente medido. Exemplos de métricas incluem: o tamanho de um produto em linhas de código; o índice *Fog* (GUNNING, 1962), que é uma medida da legibilidade de uma passagem de texto escrito; o número de defeitos relatados em um produto de software entregue, e o número de pessoas/dia requerido para desenvolver um componente de sistema.

As métricas de software podem ser métricas de controle ou métricas de previsão. Como os nomes sugerem, as primeiras suportam os processos de gerenciamento e as outras o ajudam a prever as características do software. As métricas de controle são geralmente associadas com os processos de software. Exemplos de métricas de controle ou de processos são o esforço médio e o tempo necessário para reparar os defeitos relatados. Métricas de previsão são associadas com o software em si e, por vezes, são conhecidas como 'métricas de produto'. São exemplos de métricas de previsão: a complexidade ciclomática de um módulo (discutida no Capítulo 8), o comprimento médio dos identificadores em um programa e o número de atributos e operações associadas com as classes de objeto em um projeto.

As métricas de controle e de previsão podem influenciar a tomada de decisão de gerenciamento, como mostrado na Figura 24.6. Os gerentes usam métricas de processo para decidir se devem ser feitas alterações no processo; as métricas de previsão são usadas para ajudar a estimar o esforço necessário para fazer as alterações no software. Neste capítulo, discuto principalmente as métricas de previsão, cujos valores são avaliados, analisando o código de um sistema de software. No Capítulo 26, as métricas de controle e como elas são usadas na melhoria de processos são abordadas.

Existem duas maneiras para o uso das medições de um software de sistema:

1. *Para atribuir um valor aos atributos de qualidade de sistema.* Ao medir as características dos componentes de sistema, bem como sua complexidade ciclomática e, em seguida, agregar essas medições, você pode avaliar os atributos de qualidade do sistema, como a manutenibilidade.

Figura 24.6 Medições de previsão e de controle

2. *Para identificar os componentes de sistema cuja qualidade não atingiu o padrão*. As medições podem identificar componentes individuais com características que se desviem da norma. Por exemplo, você pode medir componentes para descobrir aqueles com a mais alta complexidade. Esses são mais passíveis de conter *bugs* porque a complexidade os torna mais difíceis de entender.

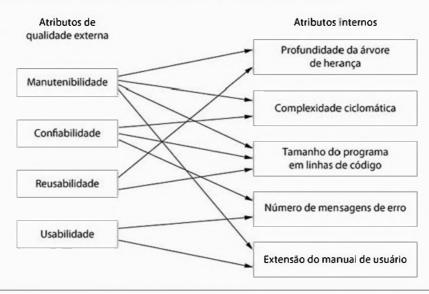
Infelizmente, é difícil fazer medições diretas de muitos dos atributos de qualidade de software mostrados na Tabela 24.1. Os atributos de qualidade como manutenibilidade, compreensibilidade e usabilidade são atributos externos relacionados com os desenvolvedores e usuários que experimentam o software. Eles são afetados por fatores subjetivos, como a experiência e a educação do usuário e, portanto, não podem ser medidos objetivamente. Para fazer um julgamento sobre esses atributos, você deve medir alguns atributos internos do software (como tamanho, complexidade etc.) e assumir que estão relacionados com as características de qualidade com as quais você se preocupa.

A Figura 24.7 mostra alguns atributos externos de qualidade dos softwares e atributos internos que poderiam, intuitivamente, ser relacionados a eles. O diagrama sugere que pode haver relacionamentos entre atributos internos e externos, mas não diz como esses atributos estão relacionados. Se a medida do atributo interno deve ser uma previsão útil de características externas de software, três condições devem ser mantidas (KITCHENHAM, 1990):

1. O atributo interno deve ser medido com precisão. Isso nem sempre é simples e pode exigir ferramentas especiais para fazer as medições.
2. Deve existir um relacionamento entre o atributo que pode ser medido e o atributo de qualidade externa de interesse. Ou seja, o valor do atributo de qualidade deve ser relacionado, de alguma forma, com o valor do atributo que pode ser medido.
3. Esse relacionamento entre os atributos internos e externos deve ser compreendido, validado e expresso em termos de uma fórmula ou modelo. A formulação de modelo envolve a identificação da forma funcional do modelo (linear, exponencial etc.) pela análise de dados coletados, identificando os parâmetros que devem ser incluídos no modelo e calibrando esses parâmetros com o uso dos dados existentes.

Os atributos internos de software, como a complexidade ciclomática de um componente, são medidos com o uso de ferramentas de software que analisam o código-fonte do software. As ferramentas *open source* disponíveis podem ser usadas para fazer essas medições. Embora a intuição sugira que poderia haver um relacionamento entre a complexidade de um componente de software e o número de falhas observadas em uso, é difícil demonstrar objetivamente que é este o caso. Para testar essa hipótese, você precisa de dados de falha de um grande número de componentes e de acesso ao código-fonte do componente para análise. Poucas empresas fizeram um compromisso a longo prazo para a coleta de dados sobre seu software, portanto, dados de falha raramente são disponibilizados para análise.

Na década de 1990, grandes empresas, como a Hewlett-Packard (GRADY, 1993), a AT&T (BARNARD e PRICE, 1994) e a Nokia (KILPI, 2001) introduziram programas de métricas. Elas fizeram medições de seus

Figura 24.7 Relacionamentos entre os atributos internos e externos de software

produtos e processos e usaram-nas em seus processos de gerenciamento de qualidade. Elas se centraram em coletar métricas de defeitos de programa, bem como em processos de verificação e validação. Offen e Jeffrey (1997) e Hall e Fenton (1997) discutem a introdução de programas de métricas na indústria com mais detalhes.

Existem poucas informações publicamente disponíveis sobre o uso atual da medição sistemática de software na indústria. Muitas empresas coletam informações sobre seu software, como o número de solicitações de mudança de requisitos ou o número de defeitos descobertos nos testes. No entanto, não está claro se, em seguida, usam essas medições sistematicamente para comparar produtos e processos de software ou avaliar o impacto das mudanças nos processos e ferramentas de software. Existem várias razões pelas quais isso é difícil:

1. É impossível quantificar o retorno sobre o investimento da introdução de um programa de métricas em organizações. Durante os últimos anos, houve melhorias significativas na qualidade de software sem o uso de métricas. Por isso é difícil justificar os custos iniciais de introdução de medição e avaliação sistemática de software.
2. Não existe um padrão para as métricas de software ou processos padronizados para medição e análise. Muitas empresas relutam em introduzir programas de medição até que estes e as ferramentas de suporte estejam disponíveis.
3. Em muitas empresas, os processos de software não são padronizados e são mal definidos e mal controlados. Além disso, existe muita variabilidade em processos da mesma empresa para que as medições sejam usadas de forma significativa.
4. Grande parte da pesquisa sobre medição e métricas de software centra-se nas métricas baseadas em códigos e processos de desenvolvimento dirigidos a planos. No entanto, cada vez mais o software é desenvolvido configurando sistemas ERP ou COTS ou usando os métodos ágeis. Não sabemos, portanto, se a pesquisa anterior é aplicável a essas técnicas de desenvolvimento de software.
5. A introdução da medição acrescenta *overhead* aos processos. Estes contradizem os objetivos dos métodos ágeis, os quais recomendam a eliminação das atividades de processos que não estão diretamente relacionadas ao desenvolvimento de programa. Portanto, as empresas que adotaram os métodos ágeis geralmente não adotam um programa de métricas.

As métricas e medições de software são as bases da engenharia de software empírica (ENDRES e ROMBACH, 2003). Essa é uma área de pesquisa em que as experiências em sistemas de software e a coleta de dados sobre projetos reais foram usadas para formar e validar hipóteses sobre os métodos e as técnicas de engenharia de software. Os pesquisadores que trabalham nessa área argumentam que podemos confiar no valor dos métodos e das técnicas de engenharia de software apenas se pudermos fornecer evidências concretas de que eles realmente oferecem os benefícios que seus inventores sugerem.

Infelizmente, mesmo quando é possível fazer medições objetivas e tirar conclusões a partir delas, isso pode não convencer os tomadores de decisões necessariamente. Em vez disso, as tomadas de decisões muitas vezes são influenciadas por fatores subjetivos, como a novidade ou a extensão em que as técnicas são de interesse para os profissionais. Portanto, penso que ainda se passarão muitos anos antes que a engenharia de software empírica tenha efeitos significativos nas práticas de engenharia de software.

24.4.1 Métricas do produto

As métricas de produto são métricas de previsão usadas para medir atributos internos de um sistema de software. O tamanho de sistema, medido em linhas de código, ou o número de métodos associados a cada classe de objeto são exemplos de métricas de produto. Contudo, como já foi explicado nesta seção, as características de software que podem ser facilmente medidas, como tamanho e complexidade ciclomática, não têm uma relação clara e consistente com atributos de qualidade tais como capacidade de compreensão e manutenibilidade. Os relacionamentos variam de acordo com os processos de desenvolvimento e tecnologia usada, bem como o tipo de sistema que está sendo desenvolvido.

As métricas de produto se dividem em duas classes:

1. Métricas dinâmicas, as quais são coletadas por meio de medições efetuadas de um programa em execução. Essas métricas podem ser coletadas durante o teste de sistema ou após o sistema estar em uso. Um exemplo pode ser o número de relatórios de *bugs* ou o tempo necessário para concluir uma computação.

2. Métricas estáticas, que são coletadas por meio de medições feitas de representações do sistema, como o projeto, o programa ou a documentação. Exemplos de métricas estáticas são o tamanho de código e o comprimento médio de identificadores usados.

Esses tipos de métrica estão relacionados com diferentes atributos de qualidade. Métricas dinâmicas ajudam a avaliar a eficiência e a confiabilidade de um programa. Métricas estáticas ajudam a avaliar a complexidade, a compreensibilidade e a manutenibilidade de um sistema ou componentes de um sistema de software.

Geralmente, existe um relacionamento claro entre as métricas dinâmicas e as características de qualidade de software. É bastante fácil medir o tempo de execução necessário para funções particulares e avaliar o tempo necessário para iniciar um sistema. Eles se relacionam diretamente com a eficiência do sistema. Da mesma forma, o número de falhas de sistema e o tipo de falha podem ser registrados e relacionados diretamente com a confiabilidade do software, conforme vimos no Capítulo 15.

Como já discutido, métricas estáticas, tais como as mostradas na Tabela 24.4, têm um relacionamento indireto com atributos de qualidade. Um grande número de métricas diferentes foi proposto e muitos experimentos tentaram derivar e validar os relacionamentos entre essas métricas e atributos, como a complexidade de sistema e manutenibilidade. Nenhum desses experimentos foi conclusivo, mas o tamanho de programa e a complexidade de controle parecem ser os mais confiáveis mecanismos de previsão de compreensibilidade, complexidade e manutenibilidade de sistema.

As métricas na Tabela 24.4 são aplicáveis a qualquer programa, mas métricas orientadas a objetos (OO) mais específicas também foram propostas. A Tabela 24.5 resume o conjunto de Chidamber e Kemerer (1994), às vezes, chamado *suite* CK, de seis métricas orientadas a objetos. Embora elas tenham sido originalmente propostas na década de 1990, ainda são as métricas OO mais amplamente usadas. Algumas ferramentas de projeto de UML coletam valores automaticamente para essas métricas quando são criados diagramas UML.

El-Amam (2001), em uma excelente revisão de métricas orientadas a objetos, discute as métricas CK e outras métricas OO e conclui que nós ainda não temos provas suficientes para compreender como essas e outras métricas orientadas a objetos se relacionam com as qualidades externas de software. Essa situação não mudou muito desde sua análise em 2001. Ainda não sabemos como usar as medições de programas orientados a objeto para tirar conclusões confiáveis sobre sua qualidade.

Tabela 24.4 Métricas estáticas de produto de software

Métrica de software	**Descrição**
Fan-in/Fan-out	*Fan-in* é a medida do número de funções ou métodos que chamam outra função ou método (digamos X). *Fan-out* é o número de funções que são chamadas pela função de X. Um valor alto para *fan-in* significa que X está fortemente acoplado ao resto do projeto e alterações em X terão repercussões extensas. Um valor alto para *fan-out* sugere que a complexidade geral do X pode ser alta por causa da complexidade da lógica de controle necessário para coordenar os componentes chamados.
Comprimento de código	Essa é uma medida do tamanho de um programa. Geralmente, quanto maior o tamanho do código de um componente, mais complexo e sujeito a erros o componente é. O comprimento de código tem mostrado ser uma das métricas mais confiáveis para prever a propensão a erros em componentes.
Complexidade ciclomática	Essa é uma medida da complexidade de controle de um programa. Essa complexidade de controle pode estar relacionada à compreensibilidade de programa. A complexidade ciclomática é discutida no Capítulo 8.
Comprimento de identificadores	Essa é uma medida do comprimento médio dos identificadores (nomes de variáveis, classes, métodos etc.) em um programa. Quanto mais longos os identificadores, mais provável que sejam significativos e, portanto, mais compreensível o programa.
Profundidade de aninhamento condicional	Essa é uma medida da profundidade de aninhamento de declarações *if* em um programa. Declarações *if* profundamente aninhadas são difíceis de entender e potencialmente sujeitas a erros.
Índice *Fog*	Essa é uma medida do comprimento médio de palavras e sentenças em documentos. Quanto maior o valor de um índice *Fog* de um documento, mais difícil a sua compreensão.

Tabela 24.5 O conjunto de métricas de CK orientadas a objetos

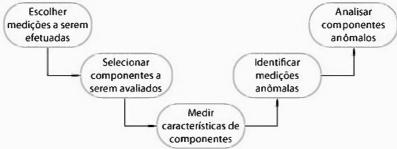
Métrica orientada a objeto	Descrição
Métodos ponderados por classe (WMC, do inglês *weighted methods per class*)	É o número de métodos em cada classe, ponderados pela complexidade de cada método. Portanto, um método simples pode ter uma complexidade de 1 e um método grande e complexo pode ter um valor muito superior. Quanto maior o valor para essa métrica, mais complexa a classe de objeto. Geralmente, os objetos complexos são mais difíceis de compreender. Eles podem não ser logicamente coesos, portanto, não podem ser reusados efetivamente como superclasses em uma árvore de herança.
Árvore de profundidade de herança (DIT, do inglês *depth Of inheritance tree*)	Representa o número de níveis discretos na árvore de herança em que as subclasses herdam atributos e operações (métodos) de superclasses. Quanto mais profunda a árvore de herança, mais complexo o projeto. Muitas classes de objeto podem precisar ser compreendidas para que as classes de objeto nas folhas da árvore sejam entendidas.
Número de filhos (NOC, do inglês *number of children*)	É uma medida do número de subclasses imediatas em uma classe. Ele mede a largura de uma hierarquia de classe, considerando que DIT mede sua profundidade. Um valor alto para NOC pode indicar um maior reúso. Isso pode significar que mais esforço deve ser dispendido na validação de classes de base por causa do número de subclasses que dependem delas.
Acoplamento entre classes de objeto (CBO, do inglês *coupling between object classes*)	Classes são acopladas quando métodos em uma classe usam métodos ou variáveis de instância definidas em uma classe diferente. CBO é uma medida de quanto acoplamento existe. Um valor alto para o CBO significa que as classes são altamente dependentes e, portanto, é mais provável que a mudança em uma classe afete outras classes do programa.
Resposta para uma classe (RFC, do inglês *response for a class*)	RFC é a medida do número de métodos que poderiam ser executados em resposta a uma mensagem recebida por um objeto dessa classe. Mais uma vez, RFC está relacionada com a complexidade. Quanto maior o valor de RFC, mais complexa é a classe e, portanto, mais provável que inclua erros.
Falta de coesão em métodos (LCOM, do inglês *lack of cohesion in methods*)	LCOM é calculada considerando os pares de métodos em uma classe. LCOM é a diferença entre o número de pares de métodos sem atributos compartilhados e o número de pares de métodos com atributos compartilhados. O valor dessa métrica tem sido amplamente discutido, e ele existe em diversas variações. Não está claro se realmente adiciona qualquer informação útil além das que já são fornecidas por outras métricas.

24.4.2 Análise de componentes de software

Um processo de medição que pode ser parte de um processo de avaliação de qualidade de software é mostrado na Figura 24.8. Cada componente de sistema pode ser analisado separadamente, usando uma variedade de métricas. Os valores dessas métricas podem ser comparados com diferentes componentes e, talvez, com dados históricos de medição obtidos em projetos anteriores. Medições anômalas, que diferem significativamente da norma, podem significar que existem problemas com a qualidade desses componentes.

Os estágios principais nesse processo de medição de componente são:

1. *Escolher medições a serem efetuadas*. As questões que a medição está destinada a responder devem ser formuladas e as medições necessárias para responder a essas questões devem ser definidas. As medições que não são

Figura 24.8 O processo de medição de produto

diretamente relevantes para essas questões não necessitam ser coletadas. O paradigma Meta-Questão-Métrica (GQM, do inglês *Goal-Question-Metric*) de Basili (BASILI e ROMBACH, 1988), discutido no Capítulo 26, é uma boa abordagem para se usar ao decidir quais dados devem ser coletados.

2. *Selecionar componentes a serem avaliados*. Você pode não precisar avaliar valores de métricas para todos os componentes de um sistema de software. Às vezes, pode selecionar um conjunto representativo de componentes para medição, que lhe permita fazer uma avaliação global da qualidade de sistema. Outras vezes, você pode se centrar nos componentes principais do sistema que estão em uso quase constante. A qualidade desses componentes é mais importante do que a qualidade de componentes raramente usados.
3. *Medir características de componentes*. Os componentes selecionados são medidos e os valores e as métricas associados, computados. Normalmente, isso envolve o processamento da representação de componente (projeto, código etc.) usando uma ferramenta automatizada de coleta de dados. Essa ferramenta pode ser especialmente escrita ou pode ser um recurso de ferramentas de projeto que já esteja em uso.
4. *Identificar medições anômalas*. Após terem sido realizadas as medições de componentes, você deve compará-los uns com os outros e com as medições anteriores que foram registradas em um banco de dados de medições. Você deve procurar valores anormalmente altos ou baixos para cada métrica, pois estes sugerem que pode haver problemas com o componente que exibe tais valores.
5. *Analisar componentes anômalos*. Ao identificar os componentes que têm valores anômalos para sua métrica escolhida, você deve examiná-los para decidir se esses valores de métricas anômalos significam que a qualidade do componente está comprometida. Um valor anômalo de métrica de complexidade (por exemplo) não significa, necessariamente, um componente de má qualidade. Pode haver alguma outra razão para o alto valor, por isso pode não significar que existam problemas de qualidade de componente.

Você sempre deve manter os dados coletados como um recurso organizacional e manter registros históricos de todos os projetos, mesmo quando dados não tenham sido usados em um determinado projeto. Uma vez estabelecido um banco de dados de medições suficientemente grande, você pode comparar a qualidade do software entre projetos e validar as relações entre os atributos de componentes internos e as características de qualidade.

24.4.3 Ambiguidade de medições

Quando você coleta dados quantitativos sobre software e processos de software, precisa analisar esses dados para entender seu significado. É fácil interpretar dados erroneamente e fazer inferências incorretas. Você não pode simplesmente olhar os dados *per se* — você também deve considerar o contexto em que eles são coletados.

Para ilustrar como os dados coletados podem ser interpretados de maneiras diferentes, considere o cenário adiante, relacionado com o número de solicitações de mudança feitas pelos usuários de um sistema:

> *Um gerente decide monitorar o número de solicitações de mudança enviadas pelos clientes com base em uma pré-suposição de que há um relacionamento entre essas solicitações de mudança e a usabilidade e adequabilidade do produto. Ele parte do princípio de que quanto maior for o número de solicitações, menos o software atenderá às necessidades do cliente.*
>
> *Tratar as solicitações de mudança de software é caro. Portanto, a organização decide modificar seu processo com o objetivo de aumentar a satisfação do cliente e, ao mesmo tempo, reduzir os custos de mudanças. A intenção é que as mudanças de processo resultarão em melhores produtos e menos solicitações de mudança.*
>
> *As mudanças de processo são iniciadas para aumentar o envolvimento do cliente no processo de projeto de software. São introduzidos testes Beta de todos os produtos e as solicitações dos clientes por modificações são incorporadas no produto entregue. Novas versões de produtos, desenvolvidas com esse processo modificado, são entregues. Em alguns casos, o número de solicitações de mudança é reduzido. Em outros, ele aumenta. O gerente fica desconcertado e considera impossível avaliar os efeitos das mudanças nos processos sobre a qualidade do produto.*

Para entender por que esse tipo de ambiguidade pode ocorrer, você deve compreender as razões pelas quais os usuários podem fazer solicitações de mudança:

1. O software não é bom o suficiente e não faz o que os clientes querem que ele faça. Portanto, eles solicitam mudanças ofereçam a funcionalidade exigida.

2. Alternativamente, o software pode ser muito bom e por isso é ampla e fortemente usado. As solicitações de mudança podem ser geradas porque existem muitos usuários de software que pensam criativamente em novas coisas que poderiam ser feitas com o software.

Portanto, aumentar o envolvimento do cliente no processo pode reduzir o número de solicitações de mudança de produtos com os quais os clientes estavam descontentes. As mudanças de processo têm sido eficazes e tornaram o software mais usável e adequado. No entanto, as mudanças de processo podem não ter funcionado e os clientes podem decidir buscar um sistema alternativo. O número de solicitações de mudança pode diminuir porque o produto perdeu parte do mercado para um produto rival e, consequentemente, existem menos usuários.

Por outro lado, as mudanças de processo poderiam trazer muitos clientes novos, felizes, que desejem participar no processo de desenvolvimento de produto. Portanto, eles geram mais solicitações de mudança. As mudanças no processo de tratamento de solicitações de mudança podem contribuir com esse aumento. Se a empresa é mais sensível aos clientes, estes podem gerar um número maior de solicitações de mudança, pois sabem que essas solicitações serão levadas a sério. Eles acreditam que suas sugestões provavelmente serão incorporadas em versões posteriores do software. Então, o número de solicitações de mudança pode ter aumentado porque os sites de beta-teste não eram típicos do maior uso do programa.

Para analisar os dados de solicitação de mudança, não basta saber o número de solicitações de mudança. Você precisa saber quem fez a solicitação, como essas pessoas usam o software e por que a solicitação foi feita. Também precisa de informações sobre fatores externos, como modificações nos procedimentos de solicitação de mudança ou alterações de mercado que podem ter efeito. Com essas informações, é possível descobrir se as mudanças de processo foram eficazes no aumento da qualidade de produto.

Isso ilustra as dificuldades de compreensão dos efeitos das mudanças e a abordagem 'científica' para esse problema é reduzir o número de fatores que possam afetar as medições feitas. No entanto, processos e produtos que estão sendo medidos não são isolados de seu ambiente. O ambiente de negócio está mudando constantemente e é impossível evitar mudanças nas práticas de trabalho só porque eles podem fazer comparações de dados inválidos. Assim, dados quantitativos sobre as atividades humanas não podem sempre ser tomados pelo valor de face. Muitas vezes, as razões pelas quais um valor medido se altera são ambíguas. Essas razões devem ser pesquisadas em detalhes antes de se tirarem conclusões a respeito de quaisquer medições efetuadas anteriormente.

- O gerenciamento de qualidade de software visa assegurar que o software tenha um pequeno número de defeitos e que se adeque aos padrões de manutenibilidade, confiabilidade, portabilidade etc. em vigor. Inclui a definição de padrões para produtos e processos e o estabelecimento de processos para verificar se tais padrões foram seguidos.
- Os padrões de software são importantes para a garantia de qualidade, pois representam uma identificação das 'melhores práticas'. Durante o desenvolvimento de software, os padrões fornecem uma base sólida para a construção de software de boa qualidade.
- Você deve documentar um conjunto de procedimentos de garantia de qualidade em um manual de qualidade organizacional. Este pode ser baseado no modelo genérico para um manual de qualidade sugerido na norma ISO 9001.
- Revisões dos entregáveis de processos de software envolvem uma equipe de pessoas que verifica se os padrões de qualidade estão sendo seguidos. Revisões são técnicas largamente usadas para avaliação de qualidade.
- Em uma inspeção de programa ou revisão em pares, uma pequena equipe verifica o código sistematicamente. Ela lê o código em detalhes procurando por possíveis erros e omissões. Em seguida, os problemas detectados são discutidos em uma reunião de revisão de código.
- A medição de software pode ser usada para coletar dados quantitativos sobre o software e o processo de software. Você pode ser capaz de usar os valores das métricas de software que são coletadas para fazer inferências sobre a qualidade de produto e de processo.
- Métricas de qualidade de produto são particularmente úteis para realçar componentes anômalos que podem ter problemas de qualidade. Em seguida, esses componentes devem ser analisados em mais detalhes.

LEITURA COMPLEMENTAR

Metrics and Models for Software Quality Engineering. 2nd edition. Discussão muito abrangente sobre as métricas de software que abrange o processo, o produto e as métricas orientadas a objetos. Ela também inclui algumas informações básicas sobre a matemática necessária para desenvolver e entender modelos baseados em medição de software. (KAN, S. H. *Metrics and Models for Software Quality Engineering*. 2. ed. Addison-Wesley, 2003.)

Software Quality Assurance: From Theory to Implementation. Um olhar excelente, atualizado sobre os princípios e as práticas de garantia de qualidade de software. Inclui uma discussão sobre normas tais como a ISO 9001. (GALIN, D. *Software Quality Assurance*: From Theory to Implementation. Addison-Wesley, 2004.)

'A Practical Approach for Quality-Driven Inspections'. Muitos artigos sobre inspecções são bastante antigos e não consideram as práticas modernas de desenvolvimento de software. Esse artigo relativamente recente descreve um método de inspeção que aborda alguns dos problemas de uso de inspeção e sugere ela pode ser usada em um ambiente de desenvolvimento moderno. (DENGER, C.; SHULL, F. *IEEE Software*, v. 24, n. 2, mar.-abr. 2007.) Disponível em: <http://dx.doi.org/10.1109/MS.2007.31>.

'Misleading Metrics and Unsound Analyses'. Excelente artigo dos principais pesquisadores de métricas que aborda as dificuldades de compreender o que as medições realmente significam. (KITCHENHAM, B.; JEFFREY, R.; CONNAUGHTON, C. *IEEE Software*, v. 24, n. 2, mar.-abr. 2007.) Disponível em: <http://dx.doi.org/10. 1109/MS.2007.49>.

'The Case for Quantitative Project Management'. Introdução para uma seção especial na revista que inclui dois outros artigos sobre gerenciamento de projetos quantitativos. Argumenta acerca de mais pesquisas em métricas e medições para melhorar o gerenciamento de projetos de software. (CURTIS, B. et al *IEEE Software*, v. 25, n. 3, mai.-jun. 2008.) Disponível em: <http://dx.doi.org/10.1109/ MS.2008.80>.

EXERCÍCIOS

24.1 Explique por que um processo de software de alta qualidade deve levar a produtos de software de alta qualidade. Discuta possíveis problemas com esse sistema de gerenciamento de qualidade.

24.2 Explique como os padrões podem ser usados para capturar a sabedoria organizacional a respeito de métodos eficazes de desenvolvimento de software. Sugira quatro tipos de conhecimentos que possam ser capturados em normas organizacionais.

24.3 Discuta a avaliação de qualidade de software de acordo com os atributos de qualidade mostrados na Tabela 24.1. Você deve considerar cada atributo e explicar como ele pode ser avaliado.

24.4 Projete um formulário eletrônico que possa ser usado para registrar comentários de revisão e que poderia ser usado para enviar por e-mail os comentários para os revisores.

24.5 Descreva rapidamente possíveis padrões que possam ser usados para:

- O uso de construções de controle em C, C# ou Java;
- Relatórios que possam ser submetidos a um projeto de formatura em uma universidade;
- O processo de fazer e aprovar mudanças de programa (ver Capítulo 26);
- O processo de comprar e instalar um novo computador.

24.6 Suponha que você trabalhe para uma organização que desenvolve produtos de banco de dados para indivíduos e empresas de pequeno porte. Essa organização está interessada na quantificação de seu desenvolvimento de software. Escreva um relatório sugerindo métricas adequadas e sugira como estas podem ser coletadas.

24.7 Explique por que inspeções de programa são uma técnica eficaz para descobrir erros em um programa. Que tipos de erros são improváveis de serem descobertos por meio de inspeções?

24.8 Explique por que as métricas de projeto são, por si só, um método inadequado de previsão de qualidade de projeto.

24.9 Explique por que é difícil validar os relacionamentos entre os atributos internos de produto, como complexidade ciclomática e atributos externos, como a manutenibilidade.

24.10 Um colega, que é um excelente programador, produz softwares com poucos defeitos, mas constantemente ignora os padrões de qualidade da organização. Como seus gerentes deveriam reagir a esse comportamento?

REFERÊNCIAS

BAMFORD, R.; DEIBLER, W. J. (Orgs.). ISO 9001:2000 for Software and Systems Providers: An Engineering Approach. Boca Raton, Flórida: CRC Press, 2003.

BARNARD, J.; PRICE, A. Managing Code Inspection Information. *IEEE Software*, v. 11, n. 2, 1994, p. 59-69.

BASILI, V. R.; ROMBACH, H. D. The TAME project: Towards Improvement-Oriented Software Environments. *IEEE Trans. on Software Eng.*, v. 14, n. 6, 1988, p. 758-773.

BOEHM, B. W.; BROWN, J. R.; KASPAR, H.; LIPOW, M.; MacLEOD, G.; MERRIT, M. *Characteristics of Software Quality*. Amsterdam: North-Holland, 1978.

CHIDAMBER, S.; KEMERER, C. A Metrics Suite for Object-Oriented Design. *IEEE Trans. on Software Eng.*, v. 20, n. 6, 1994, p. 476-493.

CROSBY, P. *Quality is Free*. Nova York: McGraw-Hill, 1979.

EL-AMAM, K. Object-oriented Metrics: A Review of Theory and Practice. National Research Council of Canada. 2001. Disponível em: <http://seg.iit.nrc.ca/English/abstracts/NRC44190.html>.

ENDRES, A.; ROMBACH, D. *Empirical Software Engineering*: A Handbook of Observations, Laws and Theories. Harlow, Reino Unido: Addison-Wesley, 2003.

FAGAN, M. E. Design and code inspections to reduce errors in program development. *IBM Systems J.*, v. 15, n. 3, 1976, p. 182-211.

______. Advances in Software Inspections. *IEEE Trans. on Software Eng.*, v. SE-12, n. 7, 1986.

GILB, T.; GRAHAM, D. *Software Inspection*. Wokingham: Addison-Wesley, 1993.

GRADY, R. B. Practical Results from Measuring Software Quality. *Comm. ACM*, v. 36, n. 11, 1993, p. 62-68.

GUNNING, R. *Techniques of Clear Writing*. Nova York: McGraw-Hill, 1962.

HALL, T.; FENTON, N. Implementing Effective Software Metrics Programs. *IEEE Software*, v. 14, n. 2, 1997, p. 55-64.

HUMPHREY, W. *Managing the Software Process*. Reading, Mass.: Addison-Wesley, 1989.

IEC. Standard IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission: Geneva, 1998.

IEEE. *IEEE Software Engineering Standards Collection on CD-ROM*. Los Alamitos, Ca.: IEEE Computer Society Press, 2003.

INCE, D. *ISO 9001 and Software Quality Assurance*. Londres: McGraw-Hill, 1994.

KILPI, T. Implementing a Software Metrics Program at Nokia. *IEEE Software*, v. 18, n. 6, 2001, p. 72-77.

KITCHENHAM, B. Measuring Software Development. In: *Software Reliability Handbook*. ROOK, P. (Org.). Amsterdam: Elsevier, 1990, p. 303-331.

McCONNELL, S. *Code Complete*: A Practical Handbook of Software Construction. 2. ed. Seattle: Microsoft Press, 2004.

MILLS, H. D.; DYER, M.; LINGER, R. Cleanroom Software Engineering. *IEEE Software*, v. 4, n. 5, 1987, p. 19-25.

OFFEN, R. J.; JEFFREY, R. Establishing Software Measurement Programs. *IEEE Software*, v. 14, n. 2, 1997, p. 45-54.

STALHANE, T.; HANSSEN, G. K. The application of ISO 9001 to agile software development. *9th International Conference on Product Focused Software Process Improvement*. PROFES 2008, Monte Porzio Catone, Itália: Springer, 2008.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 **25** 26

Gerenciamento de configurações

Objetivos

O objetivo deste capítulo é apresentar processos e ferramentas de gerenciamento de configuração. Com a leitura deste capítulo, você:

- compreenderá os processos e procedimentos envolvidos no gerenciamento de mudanças de software;
- conhecerá a funcionalidade essencial que deve ser fornecida por um sistema de gerenciamento de versões e os relacionamentos entre o gerenciamento das versões e a construção de sistemas;
- entenderá as diferenças entre uma versão de sistema e um *release* de sistema e conhecerá os estágios no processo de gerenciamento de *releases*.

Conteúdo

Os sistemas de software sempre mudam durante seu desenvolvimento e uso. *Bugs* são descobertos e precisam ser corrigidos. Os requisitos do sistema mudam e é preciso implementar essas mudanças em uma nova versão do sistema. Novas versões do hardware e novas plataformas de sistema tornam-se disponíveis e você precisa adaptar seus sistemas para trabalhar com elas. Os concorrentes introduzem novos recursos em seu sistema, aos quais você precisa corresponder. Mudanças são feitas para o software e cria-se uma nova versão de um sistema. Portanto, a maioria dos sistemas pode ser pensada como um conjunto de versões, sendo que cada uma delas necessita ser mantida e gerenciada.

O gerenciamento de configuração (CM, do inglês *configuration management*) está relacionado com as políticas, processos e ferramentas para gerenciamento de mudanças dos sistemas de software. Você precisa gerenciar os sistemas em evolução, pois é fácil perder o controle de quais mudanças e versões de componentes foram incorporadas em cada versão de sistema. As versões implementam propostas de mudanças, correções de defeitos e adaptações de hardware e sistemas operacionais diferentes. Pode haver várias versões em desenvolvimento e em uso ao mesmo tempo. Se você não tem procedimentos de gerenciamento de configuração efetivos, você pode desperdiçar esforço modificando a versão errada de um sistema, entregá-la para os clientes ou esquecer onde está armazenado o código-fonte do software para uma versão específica do sistema ou componente.

O gerenciamento de configuração é útil para projetos individuais, pois é fácil uma pessoa esquecer quais mudanças foram feitas. É essencial para projetos em equipe em que vários desenvolvedores trabalham ao mesmo tempo em um sistema de software. Às vezes, esses desenvolvedores trabalham no mesmo local, mas, cada vez mais, as equipes de desenvolvimento são distribuídas, com membros em diferentes locais pelo mundo. O uso de um sistema de gerenciamento de configuração garante que as equipes tenham acesso a informações sobre um sistema que está em desenvolvimento e não interfiram no trabalho umas das outras.

O gerenciamento de configurações de um produto de sistema de software envolve quatro atividades afins (Figura 25.1):

1. *Gerenciamento de mudanças*. Envolve manter o acompanhamento das solicitações dos clientes e desenvolvedores por mudanças no software, definir os custos e o impacto de fazer tais mudanças, bem como decidir se e quando as mudanças devem ser implementadas.
2. *Gerenciamento de versões*. Envolve manter o acompanhamento de várias versões de componentes do sistema e assegurar que as mudanças nos componentes, realizadas por diferentes desenvolvedores, não interfiram umas nas outras.
3. *Construção do sistema*. É o processo de montagem de componentes de programa, dados e bibliotecas e, em seguida, compilação e ligação destes, para criar um sistema executável.
4. *Gerenciamento de* releases. Envolve a preparação de software para o *release* externo e manter o acompanhamento das versões de sistema que foram liberadas para uso do cliente.

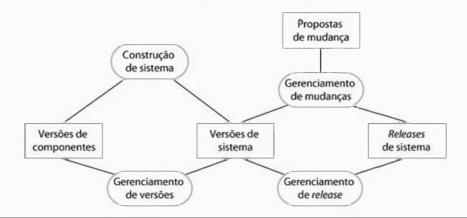
O gerenciamento de configurações envolve lidar com um grande volume de informações; dessa forma, muitas ferramentas de gerenciamento de configurações foram desenvolvidas para dar suporte a processos de CM. Elas vão desde simples ferramentas que oferecem suporte a uma tarefa única de gerenciamento de configuração, tal como acompanhamento de *bug*, até conjuntos complexos e caros de ferramentas integradas que oferecem suporte a todas as atividades de gerenciamento de configuração.

Políticas e processos de gerenciamento de configuração definem como gravar e processar propostas de mudanças de sistema, como decidir quais componentes de sistema alterar, como gerenciar diferentes versões de sistema e seus componentes e como distribuir as mudanças para os clientes. As ferramentas de gerenciamento de configuração são usadas para manter o controle das propostas de alteração, armazenar versões de componentes do sistema, construir sistemas a partir desses componentes e controlar o lançamento de versões do sistema para os clientes.

O gerenciamento de configurações, por vezes, é considerado parte do gerenciamento de qualidade de software (abordado no Capítulo 24), com o mesmo gerente tendo tanto as responsabilidades de gerenciamento de qualidade quanto as de gerenciamento das configurações. Quando uma nova versão do software é implementada, é entregue pela equipe de desenvolvimento para a equipe de garantia de qualidade (QA, do inglês *assurance quality*). A equipe de QA verifica se a qualidade de sistema é aceitável. Se assim for, torna-se um sistema controlado, o que significa que todas as mudanças no sistema precisam ser acordadas e registradas antes de serem implementadas.

A definição e a utilização de padrões de gerenciamento de configurações são essenciais para a certificação de qualidade de padrões ISO 9000 e CMM e CMMI (AHERN et al., 2001; BAMFORD e DEIBLER, 2003; PAULK et al., 1995; PEACH, 1996). Esses padrões de CM podem basear-se em normas CM genéricas que foram desenvolvidas por organismos como o IEEE. Por exemplo, a norma IEEE 828-1998 é uma norma para planos de gerenciamento de configurações. Essas normas concentram-se em processos de CM e nos documentos produzidos durante o processo de CM. Usando as normas externas como ponto de partida, as empresas desenvolvem padrões mais detalhados, especificamente adaptados às necessidades de uma empresa.

Um dos problemas do gerenciamento de configurações é que diferentes empresas falam sobre os mesmos conceitos usando termos diferentes. Existem razões históricas para tanto. Os sistemas militares de software provavelmente foram os primeiros sistemas nos quais o gernciamento de configurações foi usado e a terminologia para esses sistemas refletiu os processos e procedimentos já em vigor para o gerenciamento de configurações de hardware. Os desenvolvedores de sistemas comerciais não estavam familiarizados com os procedimentos ou terminologia militares e, muitas vezes, inventavam seus

Figura 25.1 Atividade de gerenciamento de configurações

próprios termos. Os métodos ágeis também desenvolveram novas terminologias, por vezes apresentadas deliberadamente para distinguir a abordagem ágil dos métodos tradicionais de CM. A Tabela 25.1 define a terminologia de gerenciamento de configurações que uso neste capítulo.

25.1 Gerenciamento de mudanças

A mudança é uma realidade para grandes sistemas. As necessidades e requisitos organizacionais se alteram durante a vida útil de um sistema, *bugs* precisam ser reparados e os sistemas necessitam se adaptar às mudanças em seu ambiente. Para garantir que as mudanças sejam aplicadas ao sistema de uma forma controlada, você preci-

Tabela 25.1 Terminologia de CM

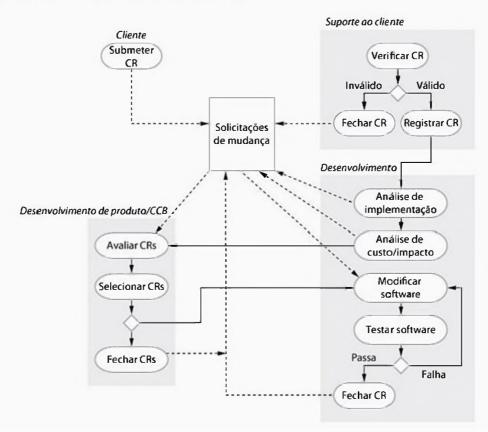
Termo	Explicação
Item de configuração ou item de configuração de software (SCI, do inglês *software configuration item*)	Qualquer coisa associada a um projeto de software (projeto, código, dados de teste, documentos etc.) que tenha sido colocado sob controle de configuração. Muitas vezes, existem diferentes versões de um item de configuração. Itens de configuração têm um nome único.
Controle de configuração	O processo de garantia de que versões de sistemas e componentes sejam registradas e mantidas para que as mudanças sejam gerenciadas e todas as versões de componentes sejam identificadas e armazenadas por todo o tempo de vida do sistema.
Versão	Uma instância de um item de configuração que difere, de alguma forma, de outras instâncias desse item. As versões sempre têm um identificador único, o qual é geralmente composto pelo nome do item de configuração mais um número de versão.
Baseline	Uma *baseline* é uma coleção de versões de componentes que compõem um sistema. As *baselines* são controladas, o que significa que as versões dos componentes que constituem o sistema não podem ser alteradas. Isso significa que deveria sempre ser possível recriar uma *baseline* a partir de seus componentes.
Codeline	Uma *codeline* é um conjunto de versões de um componente de software e outros itens de configuração dos quais esse componente depende.
Mainline	Trata-se de uma sequência de *baselines* que representam diferentes versões de um sistema.
Release	Uma versão de um sistema que foi liberada para os clientes (ou outros usuários em uma organização) para uso.
Espaço de trabalho	É uma área de trabalho privada em que o software pode ser modificado sem afetar outros desenvolvedores que possam estar usando ou modificando o software.
Branching	Trata-se da criação de uma nova *codeline* de uma versão em uma *codeline* existente. A nova *codeline* e uma *codeline* existente podem, então, ser desenvolvidas independentemente.
Merging	Trata-se da criação de uma nova versão de um componente de software, fundindo versões separadas em diferentes *codelines*. Essas *codelines* podem ter sido criadas por um *branch* anterior de uma das *codelines* envolvidas.
Construção de sistema	É a criação de uma versão de sistema executável pela compilação e ligação de versões adequadas dos componentes e bibliotecas que compõem o sistema.

sa de um conjunto de processos de gerenciamento de mudanças, apoiado por ferramentas. O gerenciamento de mudanças destina-se a garantir que a evolução do sistema seja um processo gerenciado e que seja dada prioridade às mudanças mais urgentes e efetivas.

O processo de gerenciamento de mudanças está relacionado com a análise de custos e benefícios das mudanças propostas, a aprovação dessas mudanças que valem a pena e o acompanhamento dos componentes do sistema que foram alterados. A Figura 25.2 é um modelo de um processo de gerenciamento de mudanças que mostra as principais atividades de gerenciamento. Existem muitas variantes desse processo em uso, mas, para serem eficazes, os processos de gerenciamento de mudanças sempre devem ter um meio de verificação, custeio e aprovação de mudanças. Esse processo deve entrar em vigor quando o software é desenvolvido para entrega aos clientes ou para implantação em uma organização.

O processo de gerenciamento de mudanças inicia-se quando um 'cliente' preenche e envia uma solicitação de mudança (CR, do inglês *change request*) que descreve a mudança requerida para o sistema. Pode tratar-se de um relatório de *bug*, em que são descritos os sintomas de um *bug*, ou uma solicitação para adicionar funcionalidade extra ao sistema. Algumas empresas tratam relatórios de *bugs* e novos requisitos separadamente, mas, em princípio, ambas são solicitações de mudança. As solicitações de mudança podem ser apresentadas por meio de um formulário de solicitação de alteração (CRF, do inglês *change request form*). Eu uso o termo 'cliente' para incluir qualquer *stakeholder* que não seja parte da equipe de desenvolvimento, para que as mudanças possam ser sugeridas, por exemplo, pelo departamento de marketing de uma empresa.

Formulários eletrônicos de solicitação de mudança registram informações compartilhadas entre todos os grupos envolvidos no gerenciamento de mudanças. À medida que é processada a solicitação de mudança, informação é adicionada ao CRF a fim de que se registrem as decisões tomadas em cada estágio do processo. Consequentemente, a qualquer momento, ele representa um instantâneo do estado da solicitação de mudança. Assim como registrar a mudança requerida, o CRF registra as recomendações relativas à mudança, a estimativa de custos da mudança e as datas de quando a mudança foi solicitada, aprovada, implementada e validada. O CRF também pode incluir uma seção na qual um desenvolvedor descreve como a alteração pode ser implementada.

Figura 25.2 O processo de gerenciamento de mudanças

Um exemplo de um formulário de solicitação de mudança parcialmente concluído é mostrado no Quadro 25.1. Esse é um exemplo de um tipo de CRF que pode ser usado em um grande e complexo projeto de engenharia de sistemas. Para projetos menores, eu recomendo que as solicitações de mudança sejam formalmente registradas e que o CRF se centre em descrever a mudança requerida, com menos ênfase em questões de implementação. Como um desenvolvedor de sistema, você decide como implementar essa mudança e estima o tempo necessário para tanto.

Após uma solicitação de mudança ter sido submetida, ocorre a análise de sua validade. O verificador pode ser de um cliente ou uma equipe de suporte de aplicação, ou, para solicitações internas, pode ser um membro da equipe de desenvolvimento. A verificação é necessária porque nem todas as solicitações de mudança requerem ação. Se a solicitação de mudança é um relatório de *bug*, o *bug* pode já ter sido relatado. Às vezes, o que as pessoas acreditam ser problemas são os equívocos do que se espera do sistema. Em algumas ocasiões, as pessoas solicitam recursos que já foram implementados, mas que elas não conhecem. Se qualquer um desses for verdadeiro, a solicitação de mudança é fechada e o formulário é atualizado com o motivo de encerramento. Se o caso for de uma solicitação de mudança válida, esta será registrada como uma solicitação para análise posterior.

Para solicitações de mudança válidas, o próximo estágio do processo é a avaliação e estimativa de custos de mudança. Geralmente, essa é a responsabilidade da equipe de desenvolvimento ou de manutenção, pois ela pode decidir o que está envolvido na implementação da mudança. O impacto da mudança sobre o resto do sistema deve ser verificado. Para fazer isso, você deve identificar todos os componentes afetados pela mudança. Se fazer a mudança implica na necessidade de novas mudanças em outro lugar no sistema, o custo de implementação da mudança obviamente aumentará. Em seguida, são avaliadas as mudanças requeridas para os módulos de sistema. Finalmente, o custo de fazer a mudança é estimado, levando-se em consideração os custos da mudança dos componentes relacionados.

Na sequência dessa análise, um grupo separado deve decidir se a partir de uma perspectiva de negócio, fazer a mudança no software será efetivo. Para sistemas militares e governamentais, esse grupo costuma ser chamado de Comitê de Controle de Mudanças (CCB, do inglês *Change Control Board*). Na indústria, ele pode ser chamado de algo como 'grupo de desenvolvimento de produto', responsável pela tomada de decisões sobre como um sistema de software deve evoluir. Esse grupo deve revisar e aprovar todas as solicitações de mudança, a menos que as mudanças envolvam a correção de pequenos erros em telas, páginas da Web ou documentos. Essas pequenas solicitações devem ser passadas para a equipe de desenvolvimento sem análise detalhada, pois uma análise poderia custar mais do que a implementação da mudança.

Quadro 25.1 Um formulário de solicitação de mudança parcialmente concluído

Formulário de solicitação de mudança

Projeto: SICSA/AppProcessing
Número: 23/02
Solicitante de mudança: I. Sommerville
Data: 20/jan./2009
Mudança solicitada: O ***status*** dos requerentes (rejeitados, aceitos etc.) deve ser mostrado visualmente na lista de candidatos exibida.

Analista de mudança: R. Looek
Data da análise: 25/jan./2009
Componentes afetados: ApplicantListDisplay, StatusUpdater

Componentes associados: StudentDatabase

Avaliação de mudança: Relativamente simples de implementar, alterando a cor de exibição de acordo com ***status***. Uma tabela deve ser adicionada para relacionar ***status*** a cores. Não é requerida alteração nos componentes associados.

Prioridade de mudança: Média
Implementação de mudança:
Esforço estimado: 2 horas
Data para equipe de aplicação de SGA: 28/jan./2009
Data de decisão do CCB: 30/jan./2009
Decisão: Aceitar alterar. Mudança deve ser implementada no ***Release*** 1.2
Implementador de mudança:
Data de mudança:
Data de submissão ao QA:
Decisão de QA:
Data de submissão ao CM:
Comentários:

O CCB, ou grupo de desenvolvimento de produto, considera o impacto da mudança a partir de um ponto de vista estratégico e organizacional, em vez de um ponto de vista técnico. Ele decide se a mudança em questão se justifica economicamente e prioriza, para a implementação, as mudanças aceitas. Para o grupo de desenvolvimento, as solicitações de mudança rejeitadas são fechadas e nenhuma outra ação é tomada. Fatores importantes que devem ser levados em consideração ao se decidir pela aprovação ou não de uma mudança são:

1. *As consequências de não fazer a mudança.* Ao avaliar uma solicitação de mudança, você deve considerar o que acontecerá se a mudança não for implementada. Se a mudança for associada a uma falha de sistema relatada, a gravidade da falha precisa ser levada em consideração. Se a falha de sistema causar a interrupção do sistema, isso é muito grave, e uma falha em fazer a mudança pode interromper a operação do sistema. Por outro lado, se a falha tem um efeito secundário, como cores incorretas em um *display*, então não é importante corrigir o problema rapidamente, de modo que a mudança deve ter uma prioridade baixa.
2. *Os benefícios da mudança.* A mudança é algo que beneficiará muitos usuários do sistema ou é uma proposta que beneficiará principalmente o proponente da mudança?
3. *O número de usuários afetados pela alteração.* Se apenas alguns usuários forem afetados, a mudança pode receber prioridade baixa. Na verdade, a mudança pode ser desaconselhada caso ela possa ter efeitos adversos sobre a maioria dos usuários de sistema.
4. *Os custos de se fazer a mudança.* Se fazer a mudança afetar muitos componentes de sistema (aumentando, portanto, as chances de introdução de novos *bugs*) e/ou levar muito tempo para ser implementada, então ela pode ser rejeitada, dados os elevados custos envolvidos.
5. *O ciclo de* release *de produto.* Se acaba de ser liberada uma nova versão do software para os clientes, pode fazer sentido atrasar a implementação da mudança até o próximo *release* planejado (veja a Seção 25.3).

O gerenciamento de mudanças para produtos de software (por exemplo, um produto de sistema CAD), ao invés de sistemas que são desenvolvidos especificamente para um determinado cliente, precisa ser tratado de forma ligeiramente diferente. Em produtos de software, o cliente não está diretamente envolvido nas decisões sobre a evolução de sistema, portanto, a relevância da mudança para o negócio do cliente não é um problema. As solicitações de mudança para esses produtos vêm da equipe de suporte do cliente, da equipe de marketing da empresa e dos próprios desenvolvedores. Esses pedidos podem refletir sugestões e *feedback* dos clientes ou análises do que é oferecido pelos produtos concorrentes.

A equipe de suporte do cliente pode enviar solicitações de mudança associadas com *bugs* descobertos e reportados pelos clientes após a liberação do software. Os clientes podem usar uma página da Web ou e-mail para reportar *bugs*. Em seguida, uma equipe de gerenciarmento de *bugs* verifica se os relatórios de *bug* são válidos e converte-os em solicitações formais de mudança de sistema. A equipe de marketing reúne-se com os clientes e investiga produtos competitivos. Eles podem sugerir mudanças que devem ser incluídas para facilitar as vendas de uma nova versão de um sistema para clientes antigos e atuais. Os próprios desenvolvedores de sistema podem ter algumas boas ideias sobre os novos recursos que podem ser adicionados ao sistema.

O processo de solicitação de mudança mostrado na Figura 25.2 é usado após um sistema ser liberado para os clientes. Durante o desenvolvimento, quando novas versões do sistema são criadas por meio de construções diárias (ou mais frequentes) de sistema, normalmente, um processo mais simples de gerenciamento de mudanças é usado. Os problemas e mudanças ainda devem ser registrados, mas as mudanças que afetam apenas os módulos e os componentes individuais não precisam ser avaliadas independentemente. Eles são passados diretamente para o desenvolvedor de sistema. O desenvolvedor de sistema aceita-os ou cria um exemplo explicando por que eles não são necessários. No entanto, uma autoridade independente, tal qual um arquiteto de sistema, deve avaliar e priorizar as mudanças que afetam os módulos de sistema que foram produzidos por equipes de desenvolvimento diferentes.

Em alguns métodos ágeis, tais como Extreme Programming, os clientes estão diretamente envolvidos em decidir se uma mudança deve ser implementada. Quando se propõe uma mudança para os requisitos de sistema, eles trabalham com a equipe para avaliar o impacto dessa mudança e, em seguida, decidir se a mudança deve ter prioridade sobre os recursos previstos para o próximo incremento do sistema. No entanto, as mudanças que envolvem melhorias no software são deixadas a critério dos programadores que trabalham no sistema. A refatoração, em que o software é continuamente melhorado, não é vista como um *overhead*, mas como uma parte necessária do processo de desenvolvimento.

Uma vez que a equipe de desenvolvimento muda os componentes de software, eles devem manter um registro das mudanças feitas para cada componente. Esse processo, às vezes, é chamado de 'a história de derivação de um componente'. Uma boa maneira de manter o histórico de derivação é um comentário-padrão no início do código-fonte do componente (Quadro 25.2). Esse comentário deve fazer referência à solicitação de mudança que ocasionou a mudança de software. Em seguida, você pode escrever *scripts* simples que verifiquem todos os componentes e processem os históricos de derivação para produzir relatórios de mudança de componentes. Para documentos, costumam ser mantidos registros das mudanças incorporadas em cada versão em uma página separada na frente do documento. Falarei sobre isso no capítulo disponível no site do livro sobre documentação (veja <www.pearson.com.br/sommerville>).

Geralmente, o gerenciamento de mudanças é suportado por ferramentas especializadas de software, as quais podem ser relativamente simples baseadas em Web, como o Bugzilla, que é usado para relatar problemas com muitos sistemas *open source*. Como alternativa, ferramentas mais complexas podem ser usadas para automatizar todo o processo de tratamento de solicitações de mudança, desde a proposta inicial do cliente até a aprovação de mudança.

25.2 Gerenciamento de versões

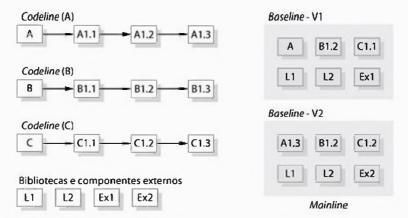
O gerenciamento de versão (VM, do inglês *version management*) é o processo de acompanhamento de diferentes versões de componentes de software ou itens de configuração e os sistemas em que esses componentes são usados. Ele também envolve a garantia de que as mudanças feitas por diferentes desenvolvedores para essas versões não interferem umas nas outras. Portanto, você pode pensar em gerenciamento de versões como o processo de gerenciamento de *codelines* e *baselines*.

A Figura 25.3 ilustra as diferenças entre *codelines* e *baselines*. Essencialmente, uma *codeline* é uma sequência de versões de código-fonte com versões posteriores na sequência derivadas de versões anteriores. Normalmente, as *codelines* são aplicadas para componentes de sistemas, havendo, assim, diferentes versões de cada componente. Uma *baseline* é uma definição de um sistema específico. A *baseline*, portanto, especifica a versão de cada componente incluída no sistema, mais uma especificação das bibliotecas usadas, arquivos de configuração etc. Na Figura 25.3, você pode ver que diferentes *baselines* usam versões diferentes dos componentes de cada *codeline*. No diagrama, eu sombreei as caixas que representam componentes na definição de *baseline* para indicar que realmente tratam de referências a componentes de uma *codeline*. A *mainline* é uma sequência de versões de sistema desenvolvidas a partir de uma *baseline* original.

As *baselines* podem ser especificadas usando-se uma linguagem de configurações que lhe permite definir quais componentes estão incluídos em uma versão de um determinado sistema. É possível especificar explicitamente a versão de um componente específico (X.1.2, por exemplo) ou, simplesmente, especificar o identificador de componente (X). Se você usar o identificador, isso significa que a versão mais recente do componente deve ser usada na *baseline*.

Quadro 25.2 Histórico de derivação

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Objeto: currentRole
// Autor: R. Looek
// Data de criação: 13/11/2009
//
// © Universidade ST. Andrews 2009
//
// Histórico de modificações
// Versão   Modificador   Data         Mudança               Razão
// 1.0      J. Jones      11/11/2009   Adicionar cabeçalho   Submetido ao CM
// 1.1      R. Looek      13/11/2009   Novo campo            Solicitação de mudança R07/02
```

Figura 25.3 *Codelines e baselines*

As *baselines* são importantes porque, muitas vezes, você precisa recriar uma versão específica de um sistema completo. Por exemplo, uma linha de produtos pode ser instanciada para que existam versões de sistema individuais para diferentes clientes. Talvez você precise recriar a versão entregue a um cliente específico, se, por exemplo, esse cliente relatar *bugs* em seu sistema, os quais precisam ser reparados.

Para dar suporte ao gerenciamento de versões, você sempre deve usar ferramentas de gerenciamento de versões (às vezes, chamadas de sistemas de controle de versões ou sistemas de controle de código-fonte). Essas ferramentas identificam, armazenam e controlam o acesso a diferentes versões de componentes. Há disponíveis muitos sistemas de gerenciamento de versões diferentes, incluindo sistemas *open source* amplamente usados como o CVS e Subversion (PILATO et al., 2004; VESPERMAN, 2003).

Normalmente, os sistemas de gerenciamento de versões fornecem uma variedade de recursos:

1. *Identificação de versão e* release. Versões gerenciadas recebem identificadores quando são submetidas ao sistema. Normalmente, esses identificadores se baseiam no nome do item de configurações (por exemplo, ButtonManager), seguido por um ou mais números. Por isso, o ButtonManager 1.3 significa a terceira versão na *codeline* 1 do componente ButtonManager. Alguns sistemas de CM também permitem a associação de atributos com versões (por exemplo, *mobile*, *smallscreen*), os quais também podem ser usados para identificação de versão. Um sistema de identificação consistente é importante porque ele simplifica o problema de definição de configurações. Ele simplifica o uso de referências de forma abreviada (por exemplo, *.V2, significando a versão 2 de todos os componentes).
2. *Gerenciamento de armazenamento.* Para reduzir o espaço requerido de armazenamento pelas várias versões de componentes que diferem apenas ligeiramente, os sistemas de gerenciamento de versões em geral fornecem recursos de gerenciamento de armazenamento. Em vez de manter uma cópia completa de cada versão, o sistema armazena uma lista de diferenças (deltas) entre uma versão e outra. Ao aplicar estas em uma versão-fonte (geralmente, a versão mais recente), uma versão de destino pode ser recriada. Esse processo é ilustrado na Figura 25.4.
3. *Registro de histórico de alterações.* Todas as mudanças feitas no código de um sistema ou componente são registadas e listadas. Em alguns sistemas, essas mudanças podem ser usadas para selecionar uma versão específica do sistema. Isso envolve marcar os componentes com palavras-chave que descrevem as mudanças feitas. Em seguida, você usa essas marcações para selecionar os componentes que serão incluídos na *baseline*.
4. *Desenvolvimento independente.* Desenvolvedores diferentes podem trabalhar no mesmo componente ao mesmo tempo. O sistema de gerenciamento de versões acompanha os componentes que tenham sido verificados para edição e garante que as mudanças feitas em um componente por diferentes desenvolvedores não interfiram.
5. *Suporte a projetos.* Um sistema de gerenciamento de versões pode apoiar o desenvolvimento de vários projetos, que compartilham componentes. Em sistemas de suporte a projetos, tal como CVS (VESPERMAN, 2003), é possível fazer *check-in* e *check-out* de todos os arquivos associados a um projeto, em vez de precisar trabalhar com um arquivo ou diretório de cada vez.

Quando foram desenvolvidos os sistemas de gerenciamento de versões, o gerenciamento de armazenamento foi uma de suas funções mais importantes. Os recursos de gerenciamento de armazenamento em um sistema de controle de versões reduzem o espaço requerido em disco para manter todas as versões de sistema. Quando uma nova versão é criada, o sistema simplesmente armazena um delta (uma lista de diferenças) entre a nova versão e a mais antiga, usada para criar essa nova versão (mostrado na parte inferior da Figura 25.4). Na Figura 25.4, as caixas sombreadas representam versões anteriores de um componente que são automaticamente recriadas a partir da versão mais recente de componente. Normalmente, os deltas são armazenados como listas de linhas alteradas e, ao aplicá-los automaticamente, uma versão de um componente pode ser criada a partir de outro. Como é mais provável que a versão mais recente de um componente será usada, a maioria dos sistemas armazena essa versão na íntegra. Os deltas definem como recriar versões anteriores de sistema.

A maior parte do processo de desenvolvimento de software é uma atividade de equipe, por isso, muitas vezes ocorrem situações em que os membros de diferentes equipes trabalham no mesmo componente ao mesmo tempo. Por exemplo, digamos que Alice está fazendo algumas mudanças em um sistema, o que envolve a mudança dos components A, B e C. Ao mesmo tempo, Bob está trabalhando em mudanças e estas requerem mudanças nos componentes X, Y e C. Portanto, Alice e Bob, estão mudando C. É importante evitar que tais mudanças interferiram umas nas outras — as mudanças em C feitas por Bob sobrescrevem as de Alice ou vice-versa.

Para apoiar o desenvolvimento independente sem interferência, sistemas de gerenciamento de versões usam o conceito de um repositório público e um espaço de trabalho privado. Os desenvolvedores realizam *check-out* de componentes de um repositório público em seu espaço de trabalho privado e podem mudá-los como quiserem em seu espaço de trabalho privado. Quando as mudanças forem concluídas, eles realizam o *check-in* de componentes para o repositório. Esse processo é ilustrado na Figura 25.5. Se duas ou mais pessoas estiverem trabalhando em um componente ao mesmo tempo, cada uma deve realizar o *check-out* de componente do repositório. Se em um componente for realizado o *check-out*, o sistema de gerenciamento de versões normalmente avisará aos outros usuários interessados em realizar o *check-out* desse componente que já foi realizado *check-out* de componente por outra pessoa. O sistema também garantirá que, quando os componentes modificados são registrados, são atribuídos identificadores de versão diferentes para diferentes versões, as quais são armazenadas separadamente.

Uma consequência do desenvolvimento independente do mesmo componente é que *codelines* podem se ramificar. Em vez de uma sequência linear de versões que refletem as mudanças para o componente ao longo do tempo, pode haver várias sequências independentes, como mostrado na Figura 25.6. Isso é normal no desenvolvimento de sistemas, em que diferentes desenvolvedores trabalham independentemente em diferentes versões do código-fonte e fazem mudanças de maneiras diferentes.

Em algum momento, pode ser necessário fundir ramificações de *codelines* para criar uma nova versão de um componente que inclui todas as mudanças realizadas. Isso também é mostrado na Figura 25.6, na qual as versões 2.1.2 e 2.3 do componente são fundidas para criar a versão 2.4. Se as mudanças feitas envolverem partes completamente diferentes do código, as versões de componente poderão ser fundidas automaticamente pelo sistema de gerenciamento de versões por meio da combinação dos deltas que se aplicam ao código. Frequentemente, existem sobreposições entre as mudanças feitas, e estas interferem umas nas outras. Um desenvolvedor deve verificar se existem conflitos e modificar as mudanças para que estas sejam compatíveis.

Figura 25.4 Gerenciamento de armazenamento usando deltas

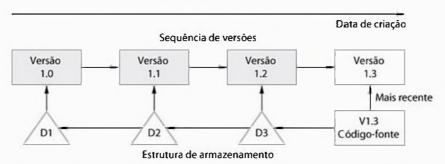



Figura 25.5 *Check-in* e *check-out* a partir de um repositório de versões

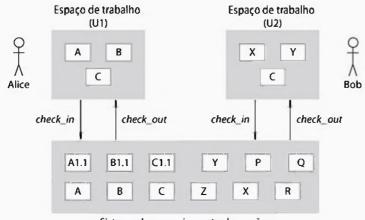
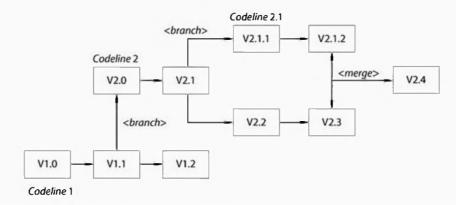
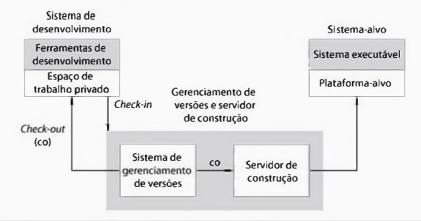



Figura 25.6 *Branching* e *merging*

25.3 Construção de sistemas

A construção de sistemas é o processo da criação de um sistema completo, executável por meio da construção e ligação dos componentes de sistema, bibliotecas externas, arquivos de configuração etc. As ferramentas de construção de sistemas e as ferramentas de gerenciamento de versões devem se comunicar na medida em que o processo de construção envolve a realização de *check-out* de versões de componentes do repositório pelo sistema de gerenciamento de versões. A descrição de configuração usada para identificar uma *baseline* também é usada pela ferramenta de construção de sistemas.

A contrução é um processo complexo e potencialmente sujeito a erros, pois pode haver três diferentes plataformas de sistemas envolvidas (Figura 25.7):

1. O sistema de desenvolvimento inclui ferramentas de desenvolvimento como compiladores, editores de código-fonte etc. Os desenvolvedores realizam o *check-out* de código do sistema de gerenciamento de versões em um espaço de trabalho privado antes de fazer as alterações no sistema. Eles podem desejar construir uma versão de um sistema para testes em seu ambiente de desenvolvimento antes da aprovação das mudanças que fizeram para o sistema de gerenciamento de versões. Isso envolve o uso de ferramentas locais de construção que usam versões de *check-out* de componentes no espaço de trabalho privado.
2. O servidor de construção é usado para construir versões definitivas e também executáveis do sistema. Dessa forma, interage estreitamente com o sistema do gerenciamento de versões. Os desenvolvedores realizam o *check-in* no sistema de gerenciamento de versões antes de este ser construído. A construção de sistema pode contar com bibliotecas externas que não estão incluídas no sistema de gerenciamento de versão.

Figura 25.7 Desenvolvimento, construção e plataformas-alvo

3. O ambiente-alvo é a plataforma em que o sistema é executado. Pode ser o mesmo tipo de computador usado para o desenvolvimento e a construção de sistemas. No entanto, para os sistemas de tempo real e embutidos, o ambiente-alvo é, geralmente, menor e mais simples do que o ambiente de desenvolvimento (por exemplo, um telefone celular). Para os sistemas de grande porte, o ambiente-alvo pode incluir bancos de dados e outros sistemas COTS que não podem ser instalados em máquinas de desenvolvimento. Em ambos os casos, não é possível construir e testar o sistema no computador desenvolvido ou no servidor de construção.

O sistema de desenvolvimento e o servidor de construção podem interagir com o sistema de gerenciamento de versões. O sistema de gerenciamento de versões pode ser hospedado tanto no servidor de construção quanto em um servidor dedicado. Para os sistemas embutidos, o ambiente de simulação pode ser instalado no ambiente de desenvolvimento para testes, ao invés de usar a plataforma real de sistema embutido. Esses simuladores podem fornecer melhor suporte de depuração do que o disponível em sistemas embutidos. No entanto, é muito difícil simular o comportamento de sistemas embutidos em todos os aspectos. Portanto, você precisa executar testes de sistema na plataforma real em que o sistema será executado, bem como o simulador de sistema.

A construção de sistemas envolve a montagem de uma grande quantidade de informações sobre o software e seu ambiente operacional. Portanto, para qualquer coisa além dos sistemas muito pequenos, sempre faz sentido usar uma ferramenta de construção automatizada para criar uma construção de sistema (Figura 25.8). Note que você não precisa apenas de arquivos de código-fonte envolvidos na construção. Você pode ter de ligá-los com bibliotecas externamente fornecidas, arquivos de dados (como um arquivo de mensagens de erro) e arquivos de configuração que definem a instalação-alvo. Talvez você precise especificar as versões do compilador e outras ferramentas de software que serão usadas na construção. A ideia é você ser capaz de construir um sistema completo com um único comando ou clique de mouse.

Existem muitas ferramentas disponíveis e um sistema de construção pode fornecer algumas características:

1. *Geração de script de construção.* Se necessário o sistema de construção deve analisar o programa que está sendo construído, identificar os componentes dependentes e gerar, automaticamente, um *script* de construção (às

Figura 25.8 A construção de sistemas

Arquivos de código-fonte
Arquivos de configuração
Testes executáveis
Arquivos de dados
Sistema automatizado de construção
Sistema-alvo executável
Bibliotecas
Compiladores e ferramentas
Resultados de testes

vezes, chamado de um arquivo de configuração). O sistema também deve apoiar a criação manual e a edição de *scripts* de construção.

2. *Integração de sistema de gerenciamento de versões.* O sistema de construção deve realizar o *check-out* das versões requeridas de componentes do sistema de gerenciamento de versões.
3. *Recompilação mínima.* O sistema de construção deve definir se o código-fonte precisa ser recompilado e configurar as compilações, caso seja necessário.
4. *Criação de sistemas executáveis.* O sistema de construção deve ligar os arquivos de código compilado uns aos outros e com outros arquivos requeridos, tais como bibliotecas e arquivos de configuração para criar um sistema executável.
5. *Automação de testes.* Alguns sistemas de construção podem executar automaticamente testes automatizados usando ferramentas de automação de testes como JUnit. Estas verificam se a construção não foi 'quebrada' pelas mudanças.
6. *Emissão de relatórios.* O sistema de construção deve fornecer relatórios sobre o sucesso ou a falha da construção, bem como os testes que foram executados.
7. *Geração de documentação.* O sistema de construção pode ser capaz de gerar notas de *release* sobre a construção e páginas de ajuda de sistema.

O *script* de construção é uma definição do sistema a ser construído. Ele inclui informações sobre os componentes e suas dependências, além das versões das ferramentas usadas para compilar e ligar o sistema. O *script* de construção inclui a especificação de configuração e, assim, a linguagem de *script* usada é a mesma que a linguagem de descrição de configuração. A linguagem da configuração inclui construções para descrever os componentes de sistema a serem incluídos na construção e suas dependências.

Como a compilação é um processo computacionalmente intensivo, as ferramentas de apoio à construção de sistema geralmente são projetadas para minimizar a quantidade de compilação requerida. Elas fazem isso verificando se uma versão compilada de um componente está disponível. Se assim for, não existe a necessidade de recompilar esse componente. Por isso, deve haver uma forma não ambígua de ligar o código-fonte de um componente com seu código-objeto equivalente.

A maneira como isso é feito é associar uma assinatura única a cada arquivo em que um código-fonte é armazenado. O código-objeto correspondente, que foi compilado do código-fonte, tem uma assinatura relacionada. A assinatura identifica cada versão do código-fonte e é alterada quando o código-fonte é editado. Comparando as assinaturas nos arquivos de código-fonte e de código do objeto, é possível concluir se o componente de código-fonte foi usado para gerar o componente de código-objeto.

Existem dois tipos de assinaturas que podem ser usadas:

1. Timestamps *de modificação.* A assinatura no arquivo de código-fonte indica a hora e a data em que o arquivo foi modificado. Se o arquivo de código-fonte de um componente for modificado após o arquivo de código-objeto relacionado, o sistema pressupõe que seja necessária uma reconstrução para criar um novo arquivo de código-objeto.

   Por exemplo, digamos que componentes Comp.java e Comp.class possuem assinaturas de modificação de 17:03:05:02:14:2009 e 16:34:25:02:12:2009, respectivamente. Isso significa que o código Java foi modificado em três minutos e cinco segundos após 5 no dia 14 de fevereiro de 2009 e a versão compilada foi modificada em 34 minutos e 25 segundos após 4 no dia 12 de fevereiro de 2009. Nesse caso, o sistema recompilaria automaticamente Comp.java, pois a versão compilada não inclui as alterações feitas ao código-fonte desde 12 de fevereiro.
2. Checksums *de código-fonte.* A assinatura no arquivo do código-fonte é um *checksum* calculado a partir da data do arquivo. Uma função de *checksum* calcula um número exclusivo usando o texto-fonte como entrada. Se você alterar o código-fonte (até mesmo em um caractere), gerará um *checksum* diferente. Portanto, esteja certo de que arquivos de código-fonte com diferentes *checksums* são realmente diferentes. O *checksum* é atribuído ao código-fonte antes da construção e identifica unicamente o arquivo-fonte. O arquivo do código-objeto gerado é marcado com a assinatura de *checksum*. Se não houver um arquivo de código-objeto com a mesma assinatura que o arquivo de código-fonte que deve ser incluído em um sistema, é necessária a recompilação do código-fonte.

Como os arquivos de código-objeto não são normalmente versionados, a primeira abordagem significa que somente o arquivo de código-objeto mais recentemente compilado será mantido no sistema. Ele é relacionado, em geral, ao arquivo de código-fonte pelo nome (ou seja, ele tem o mesmo nome que o arquivo de código-

-fonte, mas com um sufixo diferente). Portanto, o arquivo-fonte Comp.java pode gerar o arquivo-objeto Comp.class. Como os arquivos-fonte e objeto são ligados por nome, em vez de uma assinatura de arquivo-fonte explícita, não costuma ser possível construir versões diferentes de um componente de código-fonte no mesmo diretório ao mesmo tempo, pois estas gerariam arquivos de objeto com o mesmo nome.

A abordagem de *checksum* tem a vantagem de permitir que várias versões diferentes do código-objeto de um componente possam ser mantidas ao mesmo tempo. A assinatura, em vez do nome do arquivo, é a ligação entre o código-fonte e o objeto. Os arquivos de código-fonte e de código-objeto têm a mesma assinatura. Portanto, quando você recompilar um componente, ele não sobrescreverá o código-objeto, como normalmente seria o caso se o *timestamp* fosse usado. Em vez disso, ele gera um novo arquivo de código-objeto e o marca o com a assinatura de código-fonte. A compilação paralela é possível e diferentes versões de um componente podem ser compiladas ao mesmo tempo.

Os métodos ágeis recomendam que as construções de sistema muito frequentes devem ser feitas com testes automatizados (às vezes, chamados testes fumaça) para descobrir os problemas de software. As construções frequentes podem ser parte de um processo de integração contínua, como mostrado na Figura 25.9. De acordo com a noção de métodos ágeis de fazer muitas pequenas mudanças, a integração contínua envolve a frequente reconstrução da *mainline*, após terem sido feitas pequenas alterações no código-fonte. As etapas na integração contínua são:

1. Realizar *check-out* do sistema de *mainline* do sistema de gerenciamento de versões no espaço de trabalho privado do desenvolvedor.
2. Construir o sistema e executar testes automatizados para assegurar que o sistema construído passe em todos os testes. Caso contrário, a construção é interrompida e você deve informar quem realizou o último *check-in* de sistema de *baseline*. Eles são responsáveis por reparar o problema.
3. Fazer as mudanças para os componentes de sistema.
4. Construir o sistema no espaço de trabalho privado e executar novamente os testes de sistema. Se os testes falharem, continue a editar.
5. Uma vez que o sistema tenha passado nos testes, verificar isso no sistema construído, mas não aprovar como uma nova *baseline* de sistema.
6. Construir o sistema no servidor de construção e executar os testes. Você precisará fazer isso para o caso de outras pessoas terem modificado os componentes depois que você tenha feito o *check-out* do sistema. Se esse for o caso, conferir os componentes que falharam e editá-los, de maneira que os testes passem em seu espaço de trabalho privado.
7. Se o sistema passar em seus testes sobre o sistema de construção, então, aprovar as mudanças feitas como uma nova *baseline* no *mainline* de sistema.

Figura 25.9 Integração contínua

O argumento para a integração contínua é que ele permite que os problemas causados pelas interações entre diferentes desenvolvedores sejam detectados e reparados tão logo isso seja possível. O sistema mais recente na *mainline* é o sistema de trabalho definitivo. No entanto, embora a integração contínua seja uma boa ideia, nem sempre é possível implementar essa abordagem para a construção de sistemas. As razões para isso são:

1. Se o sistema for muito grande, pode levar muito tempo para ser construído e testado. É impraticável construir esse sistema várias vezes ao dia.
2. Se a plataforma de desenvolvimento é diferente da plataforma-alvo, pode não ser possível executar testes de sistema no espaço de trabalho privado do desenvolvedor. Pode haver diferenças de hardware, sistema operacional ou software instalado. Portanto, é necessário mais tempo para testar o sistema.

Para sistemas de grande porte ou sistemas em que a plataforma de execução não seja a mesma que a plataforma de desenvolvimento, a integração contínua pode ser impraticável. Nessas circunstâncias, um sistema de construção diária pode ser usado. As características disso são as seguintes:

1. A organização de desenvolvimento define um tempo de entrega (digamos, às 14 horas) para os componentes de sistema. Se os desenvolvedores tiverem novas versões de componentes que estejam escrevendo, nessa hora eles devem entregá-las. Os componentes podem estar incompletos, mas devem fornecer alguma funcionalidade básica que pode ser testada.
2. Uma nova versão do sistema é construída a partir desses componentes por sua compilação e ligação para formar um sistema completo.
3. Em seguida, esse sistema é entregue para a equipe de testes, que realiza um conjunto de testes de sistema predefinidos. Ao mesmo tempo, os desenvolvedores ainda estão trabalhando em seus componentes, adicionando funcionalidade e reparando defeitos descobertos em testes anteriores.
4. Defeitos que são descobertos durante os testes de sistema são documentados e retornam para os desenvolvedores de sistema. Eles reparam tais defeitos em uma versão posterior do componente.

As vantagens do uso frequente de construções de software são que as chances de encontrar problemas resultantes de interações de componentes no início do processo aumentam. A construção frequente incentiva os testes unitários de componentes. Psicologicamente, os desenvolvedores são colocados sob pressão para não 'quebrarem a construção', ou seja, eles tentam evitar verificar versões de componentes que causem falhas no sistema como um todo. Portanto, são relutantes em entregar novas versões de componentes que não foram devidamente testadas. Consequentemente, menos tempo é gasto durante o sistema de teste descobrindo e lidando com defeitos de software que poderiam ter sido encontrados pelo desenvolvedor.

25.4 Gerenciamento de *releases*

Um *release* de sistema é uma versão de um sistema de software distribuída aos clientes. Para softwares de mercado de massa, é normalmente possível identificar dois tipos de *releases*, chamados *releases* principais, que fornecem nova e significativa funcionalidade, e *releases* menores, que reparam *bugs* e corrigem problemas de clientes que foram relatados. Por exemplo, este livro está sendo escrito em um computador Mac da Apple, no qual o sistema operacional é OS 10.5.8. Isso significa *release* menor 8 do *release* maior 5 do OS 10. Os *releases* principais são economicamente muito importantes para o fornecedor de software, pois os clientes precisam pagar por eles. Em geral, os *releases* menores são distribuídos gratuitamente.

Para softwares customizados ou linhas de produtos de software, o gerenciamento de *releases* de sistema é um processo complexo. Os *releases* especiais do sistema podem precisar ser produzidos para cada cliente e os clientes individuais podem estar executando vários *releases* diferentes do sistema ao mesmo tempo. Isso significa que uma empresa de software vendendo um produto de software especializado pode precisar gerenciar dezenas ou até centenas de diferentes *releases* desse produto. Seus sistemas e processos de gerenciamento de configurações precisam ser projetados para fornecer informações sobre qual *release* do sistema cada cliente tem e o relacionamento entre os *releases* e as versões de sistema. No caso de algum problema, pode ser necessário reproduzir exatamente o software entregue para um determinado cliente.

Assim, quando um *release* de sistema é produzido, deve-se documentar para se garantir que ele possa ser recriado no futuro. Isso é particularmente importante para sistemas embutidos, customizados e de longa vida, tais como aqueles que controlam máquinas complexas. Os clientes podem usar um único *release* desses sistemas por

muitos anos e podem exigir mudanças específicas para um sistema de software específico muito após a data de *release* original.

Para documentar um *release*, você deve gravar as versões específicas dos componentes de código-fonte que foram usadas para criar o código executável. Você deve manter cópias dos arquivos de código-fonte executáveis correspondentes e todos os dados e arquivos de configuração. Você também deve gravar as versões do sistema operacional, bibliotecas, compiladores e outras ferramentas usadas para construir o software. Estas podem ser necessárias para construir exatamente o mesmo sistema em uma data posterior. Isso pode significar que você precisa armazenar cópias de plataforma de software e as ferramentas usadas para criar o sistema no sistema de gerenciamento de versões junto com o código-fonte do sistema-alvo.

Preparar e distribuir um *release* de sistema é um processo caro, sobretudo para produtos de software de mercado de massa. Assim como os trabalhos técnicos envolvidos na criação de um *release* de distribuição, a publicidade e o material de propaganda precisam estar preparados e estratégias de marketing devem ser criadas para convencer os clientes a comprarem o novo *release* do sistema. O calendário de *release* deve ser pensado com cuidado. Se os *releases* são muito frequentes ou requerem atualizações de hardware, os clientes podem optar por não mudar para o novo *release*, especialmente se precisarem pagar por ele. Se os *releases* de sistema não forem muito frequentes, é possível perder sua fatia de mercado, pois os clientes podem mudar para sistemas alternativos.

Os vários fatores técnicos e organizacionais que devem ser levados em consideração ao se decidir quando lançar uma nova versão de um sistema são mostrados na Tabela 25.2.

Um *release* de sistema não é apenas o código executável do sistema. Ele também pode incluir:

- arquivos de configuração definindo como o *release* deve ser configurado para instalações particulares;
- arquivos de dados, tais como arquivos de mensagens de erro, que são necessários para uma operação bem-sucedida do sistema;
- um programa de instalação, o qual é usado para ajudar a instalar o sistema no hardware-alvo;
- documentação eletrônica e em papel, que descrevem o sistema;
- empacotamento e publicidade associada, projetadas para esse lançamento.

A criação de um *release* é o processo de criação da coleção de arquivos e documentação que inclui todos os componentes do *release* de sistema. O código executável de programas e todos os arquivos de dados associados devem ser identificados no sistema de gerenciamento de versões e marcados com o identificador de versão. As des-

Tabela 25.2 Fatores que influenciam o planejamento de *release* de sistema

Fator	**Descrição**
Qualidade técnica do sistema	Caso sejam relatados defeitos graves de sistema, que afetem a maneira como muitos clientes o usam, pode ser necessário emitir um *release* de reparação de defeitos. Pequenos defeitos de sistema podem ser reparados mediante a emissão de *patches* (normalmente distribuídos pela Internet) que podem ser aplicados no *release* atual do sistema.
Mudanças de plataforma	Talvez você precise criar um novo *release* de uma aplicação de software quando uma nova versão da plataforma do sistema operacional for lançada.
Quinta lei de Lehman (ver Capítulo 9)	Essa 'lei' sugere que se você adicionar nova funcionalidade a um sistema, você também introduzirá *bugs* que limitarão a quantidade de funcionalidade que pode ser incluída no próximo *release*. Portanto, um *release* de sistema com funcionalidade nova e significativa pode ser seguido por um *release* que se concentra em reparar os problemas e melhorar o desempenho.
Concorrência	Para software de mercado de massa, um novo *release* de sistema pode ser necessário porque um produto concorrente introduziu novos recursos e a fatia de mercado pode ser perdida caso estes não sejam fornecidos aos clientes existentes.
Requisitos de marketing	O departamento de marketing de uma organização pode ter feito um compromisso para *releases* estarem disponíveis em uma determinada data.
Propostas de mudança de cliente	Para sistemas customizados, os clientes podem ter feito e pago por um conjunto específico de propostas de mudanças de sistema e eles esperam um *release* de sistema assim que estas sejam implementadas.

crições de configuração podem precisar ser escritas para diferentes hardwares e sistemas operacionais, e instruções preparadas para clientes que precisam configurar seus próprios sistemas. Se forem distribuídos manuais legíveis por máquina, as cópias eletrônicas deverão ser armazenadas com o software. Pode ser necessário escrever os ***scripts*** para o programa de instalação. Finalmente, quando toda a informação estiver disponível, uma imagem mestre executável do software deve ser preparada e entregue para distribuição para os clientes ou pontos de venda.

Ao planejar a instalação de novos *releases* de sistema, você não pode presumir que os clientes sempre instalarão novos *releases* de sistema. Alguns usuários de sistema podem estar felizes com um sistema existente. Eles podem considerar que não vale a pena o custo da mudança para um novo *release*. Portanto, novos *releases* de sistema não podem contar com a instalação de *releases* anteriores. Para ilustrar esse problema, considere o seguinte cenário:

1. O *release* 1 de um sistema é distribuído e colocado em uso.
2. O *release* 2 requer a instalação de novos arquivos de dados, mas alguns clientes não precisam dos recursos do *release* 2 e, assim, permanecem com o *release* 3. O *release* 3 requer os arquivos de dados instalados no *release* 2 e não tem novos arquivos de dados próprio.

O distribuidor de software não pode presumir que os arquivos necessários para o *release* 3 já tenham sido instalados em todos os sites. Alguns sites podem ir diretamente do *release* 1 para o *release* 3, ignorando o *release* 2. Alguns sites podem ter modificado os arquivos de dados associados ao *release* 2 a fim de refletirem as circunstâncias locais. Portanto, os arquivos de dados devem ser distribuídos e instalados com o *release* 3 do sistema.

Os custos de marketing e empacotamento associados aos novos *releases* dos produtos de software são elevados para que, assim, os fornecedores de produtos apenas criem novos *releases* para novas plataformas ou adicionem funcionalidade nova e significativa. Então, eles cobram os usuários por esse novo software. Quando problemas são descobertos em um *release* existente, os fornecedores de software fazem *patches*, para reparar o software existente disponível, em um website para que eles possam ser baixados pelos clientes.

O problema com o uso de *patches* que podem ser baixados é que muitos clientes poderão nunca descobrir a existência desses reparos de problema e talvez não entendam por que eles devem ser instalados. Em vez disso, podem continuar usando seu sistema existente, com problemas, com os consequentes riscos para seus negócios. Em algumas situações, em que o *patch* é projetado para reparar brechas de proteção, os riscos de não conseguir instalar o *patch* podem significar que o negócio é suscetível a ataques externos. Para evitar esses problemas, os fornecedores de software de mercado de massa, tais como Adobe, Apple e Microsoft, geralmente implementam atualizações automáticas em que os sistemas são atualizados sempre que um novo *release* menor se torna disponível. No entanto, normalmente, isso não funciona para sistemas customizados porque esses sistemas não existem em uma versão-padrão para todos os clientes.

PONTOS IMPORTANTES

- O gerenciamento de configurações é o gerenciamento de um sistema de software em constante evolução. Durante a manutenção de um sistema, uma equipe de CM é responsável por garantir que as mudanças sejam incorporadas no sistema de uma forma controlada e que os registros sejam mantidos com os detalhes das mudanças que foram implementadas.
- Os principais processos de gerenciamento de configurações estão interessados no gerenciamento de mudanças, no gerenciamento de versões, na construção de sistema e no gerenciamento de *releases*. Ferramentas de software estão disponíveis para apoiar todos esses processos.
- O gerenciamento de mudanças envolve avaliar propostas de mudanças dos clientes de sistema e outros *stakeholders* e decidir se é efetivo implementá-las em um novo *release* de um sistema.
- O gerenciamento de versões envolve o acompanhamento das diferentes versões dos componentes de software criadas à medida que as mudanças são feitas.
- A construção de sistemas é o processo de montagem de componentes de sistema em um programa executável para executar em um sistema de computador-alvo.
- O software deve ser reconstruído e testado frequentemente, imediatamente após a construção de uma nova versão. Isso facilita a detecção de *bugs* e problemas introduzidos desde a última construção.
- Os *releases* de sistema incluem códigos executáveis, arquivos de dados, arquivos de configuração e documentação. O gerenciamento de versões envolve tomar decisões sobre as datas de *release* de sistema, preparar todas as informações para distribuição e documentar cada *release* de sistema.

LEITURA COMPLEMENTAR

Configuration Management Principles and Practice. Esse livro, muito completo, abrange padrões e abordagens tradicionais de CM, além das abordagens de CM mais adequadas para os processos modernos, tais como o desenvolvimento ágil de software. (HASS, A. M. J. *Configuration Management Principles and Practice.* Addison-Wesley, 2002.)

Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Um livro relativamente curto, de fácil leitura e que dá bons conselhos práticos sobre as práticas de gerenciamento de configurações, especialmente para os métodos ágeis de desenvolvimento. (BERCZUK, S. P.; APPLETON, B. *Software Configuration Management Patterns*: Effective Teamwork, Practical Integration. Addison-Wesley, 2003.)

'High-level Best Practices in Software Configuration Management'. Esse artigo, disponível na Internet, escrito pelo pessoal de um fornecedor de ferramentas CM, é uma excelente introdução às boas práticas no gerenciamento de configurações de software. (WINGERD, L.; SEIWALD, C. 2006.) Disponível em: <http://www.perforce.com/perforce/papers/ bestpractices.html>.

'Agile Configuration Management for Large Organizations'. Esse artigo, disponível na Internet, descreve as práticas de gerenciamento de configurações que podem ser usadas em processos de desenvolvimento ágil, com ênfase em como estes podem ser escalados para empresas e projetos de grande porte. (SCHUH, P. 2007.) Disponível em: <http://www.ibm.com/developerworks/rational/library/mar07/schuh/index.html>.

EXERCÍCIOS

25.1 Sugira cinco problemas que podem surgir caso uma empresa não desenvolva políticas e processos efetivos de gerenciamento de configurações.

25.2 Quais são os benefícios de se usar um formulário de solicitação de mudança como documento central no processo de gerenciamento de mudanças?

25.3 Descreva seis características essenciais que deveriam ser incluídas em uma ferramenta de suporte aos processos de gerenciamento de mudanças.

25.4 Explique por que é essencial que cada versão de um componente seja unicamente identificada. Comente sobre os problemas do uso de um esquema que é simplesmente baseado em identificação de versões baseada em números de versão.

25.5 Imagine uma situação em que dois desenvolvedores estejam modificando três diferentes componentes de software simultaneamente. Quais dificuldades podem surgir ao se tentar fundir as mudanças que eles fizeram?

25.6 Cada vez mais, os softwares estão sendo desenvolvidos por equipes cujos membros trabalham em locais diferentes. Sugira recursos em um sistema de gerenciamento de versões que podem ser necessários para apoiar esse desenvolvimento distribuído de software.

25.7 Descreva as dificuldades que podem surgir durante a construção de um sistema a partir de seus componentes. Que problemas específicos podem ocorrer quando um sistema é construído em um computador *host* para alguma máquina-alvo?

25.8 Com referência à construção de sistemas, explique por que, às vezes, devem ser mantidos computadores obsoletos nos quais foram desenvolvidos grande sistemas de software.

25.9 Um problema comum de construção de sistemas ocorre quando nomes de arquivos físicos são incorporados ao código de sistema e a estrutura de arquivos implicada nesses nomes é diferente da máquina-alvo. Escreva um conjunto de diretrizes do programador que o ajude a evitar esse e outros problemas de construção de sistemas que você possa pensar.

25.10 Descreva cinco fatores que devem ser levados em consideração pelos engenheiros durante o processo de construção de um *release* de um grande sistema de software.

AHERN, D. M.; CLOUSE, A.; TURNER, R. *CMMI Distilled*. Reading, Mass.: Addison-Wesley, 2001.

BAMFORD, R.; DEIBLER, W. J. ISO 9001:2000 for Software and Systems Providers: An Engineering Approach. Boca Raton, FL: CRC Press, 2003.

PAULK, M. C.; WEBER, C. V.; CURTIS, B.; CHRISSIS, M. B. *The Capability Maturity Model*: Guidelines for Improving the Software Process. Reading, Mass.: Addison-Wesley, 1995.

PEACH, R. W. *The ISO 9000 Handbook*. 3. ed. Nova York: Irwin Professional Pub., 1996.

PILATO, C. M.; COLLINS-SUSSMAN, B.; FITZPATRICK, B. W. *Version Control with Subversion*. Sebastopol, Calif.: O'Reilly Media Inc., 2004.

VESPERMAN, J. *Essential CVS*. Sebastopol, Calif.: O'Reilly and Associates, 2003.

CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 **26**

Melhoria de processos

Objetivos

O objetivo deste capítulo é introduzir o leitor ao tema da melhoria de processos de software como uma maneira de aumentar a qualidade de software e reduzir os custos de desenvolvimento. Com a leitura deste capítulo, você:

- compreenderá a lógica para a melhoria de processos de software como um meio de melhorar a qualidade de produtos e a eficiência e eficácia dos processos de software;
- compreenderá os princípios da melhoria de processos de software e o processo cíclico de melhoria de processo;
- saberá como a abordagem de Meta-Pergunta-Métrica (GQM, do inglês *Goal-Question-Metric*) pode ser usada para guiar a medição de processo;
- terá sido apresentado às ideias de capacidade de processo e maturidade de processo, bem como à forma geral dos modelos CMMI do SEI para melhoria de processo.

Conteúdo

Atualmente, existe uma procura constante da indústria por softwares mais baratos e melhores, os quais precisam ser entregues em *deadlines* cada vez mais rigorosos. Consequentemente, muitas empresas de software voltaram-se para a melhoria de processos de software como uma forma de melhorar a qualidade de seu software, reduzindo custos ou acelerando seus processos de desenvolvimento. A melhoria de processos implica a compreensão dos processos existentes e sua mudança para aumentar a qualidade de produtos e/ou reduzir custos e o tempo de desenvolvimento.

São usadas duas abordagens bastante diferentes para a melhoria e a mudança de processos:

1. A abordagem de maturidade de processo, a qual se centra em melhorar o gerenciamento de processos e projetos e em introduzir boas práticas de engenharia de software em uma organização. O nível de maturidade de processos reflete o grau em que as boas práticas técnicas e de gerenciamento foram adotadas nos processos de desenvolvimento de software da organização. Os principais objetivos dessa abordagem são produtos de melhor qualidade e previsibilidade de processo.
2. A abordagem ágil, a qual se centra no desenvolvimento iterativo e na redução de *overheads* gerais no processo de software. As principais características dos métodos ágeis são a entrega rápida de funcionalidade e a capacidade de resposta às mudanças de requisitos de cliente.

Geralmente, os adeptos de uma dessas abordagens são céticos em relação aos benefícios da outra. A abordagem de maturidade de processo está enraizada no desenvolvimento dirigido a planos e em geral requer aumento de '*overhead*',

no sentido de que são introduzidas atividades que não são diretamente relevantes para a programação. As abordagens ágeis concentram-se no código que está sendo desenvolvido e minimizam, deliberadamente, a formalidade e a documentação.

No Capítulo 3, e em outras partes do livro, eu discuto os métodos ágeis; então, neste capítulo eu me concentro no gerenciamento de processos e na melhoria de processos baseada em maturidade. Isso não significa que eu prefiro essa abordagem aos métodos ágeis. Na verdade, eu estou convencido de que, para projetos de pequeno e médio portes, a adoção de práticas ágeis provavelmente seja a estratégia mais efetiva de melhoria de processos. No entanto, para grandes sistemas, sistemas críticos e sistemas que envolvem os desenvolvedores em diferentes empresas, muitas vezes as questões de gerenciamento são as razões pelas quais projetos têm problemas. Para as empresas cujo negócio é a engenharia de sistemas grandes e complexos, deve-se considerar uma abordagem centrada na maturidade para melhoria de processos.

Conforme discutido no Capítulo 24, o processo de desenvolvimento usado para criar um sistema de software influencia a qualidade desse sistema. Por isso, muitas pessoas acreditam que melhorar o processo de desenvolvimento de software acarreta melhorias na qualidade de software. Essa noção de melhoria de processos é uma invenção criativa do engenheiro norte-americano W. E. Deming, que trabalhou com a indústria japonesa após a Segunda Guerra Mundial para ajudar na melhoria de qualidade. A indústria japonesa, por muitos anos, tem-se mostrado comprometida com a melhoria contínua de processos, o que levou à reconhecida alta qualidade dos bens manufaturados japoneses.

Deming (e outros) introduziram a ideia de controle estatístico de qualidade, a qual se baseia em medir o número de defeitos de produto e relacionar esses defeitos ao processo. O objetivo é reduzir o número de defeitos de produto, analisar e modificar o processo para que as chances de introdução de defeitos sejam reduzidas e a detecção de defeitos seja melhorada. Uma vez que se tenha conseguido uma baixa contagem de defeitos, o processo é padronizado e, em seguida, um novo ciclo de melhorias tem início.

Humphrey (1988), em seu livro seminal sobre gerenciamento de processos, argumenta que as mesmas técnicas podem ser aplicadas na engenharia de software. Ele afirma:

> *W. E. Deming, em seu trabalho com a indústria japonesa após a Segunda Guerra Mundial, aplicou na indústria seus conceitos do controle estatístico de processos. Embora existam diferenças importantes, esses conceitos são tão aplicáveis ao software quanto para automóveis, máquinas fotográficas, relógios de pulso e aço.*

Embora existam semelhanças claras, não concordo com Humphrey que os resultados da engenharia de manufatura possam ser facilmente transferidos para a engenharia de software. Nos casos em que a manufatura está envolvida, a relação processo/produto é óbvia. A manufatura geralmente envolve a criação de ferramentas automatizadas e processos de verificação de produtos. Se alguém comete um erro ao calibrar uma máquina, isso afetará todos os produtos produzidos por essa máquina. Evitar erros na configuração de máquinas e introduzir processos de verificação mais eficazes melhoram claramente a qualidade de produto.

Esse relacionamento entre qualidade de produto e qualidade de processo é menos óbvio quando o produto é intangível e dependente, em certa medida, de processos intelectuais que não podem ser automatizados. A qualidade de software não é influenciada por seu processo de manufatura, mas por seu processo de projeto, no qual as habilidades e a experiência das pessoas são significativas. Em alguns casos, o processo usado pode ser o determinante mais significativo da qualidade de produto. No entanto, para aplicações inovadoras, as pessoas envolvidas no processo têm mais influência sobre a qualidade do que o processo usado.

Para produtos de software, ou quaisquer outros produtos intelectuais, tais como livros ou filmes em que a qualidade do produto depende de seu projeto, existem quatro importantes fatores que afetam a qualidade de produto. Eles são mostrados na Figura 26.1.

A influência de cada um desses fatores depende do tamanho e do tipo de projeto. Para sistemas muito grandes que incluem subsistemas distintos, desenvolvidos por equipes que podem estar trabalhando em locais diferentes, o principal fator, que afeta a qualidade de produto, é o processo de software. Os principais problemas com grandes projetos são a integração, o gerenciamento de projetos e as comunicações. Geralmente, existe uma mistura de habilidades e experiência nos membros de equipe e, porque o processo de desenvolvimento geralmente ocorre ao longo de vários anos, a equipe de desenvolvimento é volátil. Ela pode mudar completamente ao longo da vida do projeto.

No entanto, para pequenos projetos, em que haja apenas alguns membros de equipe, a qualidade da equipe de desenvolvimento é mais importante do que o processo de desenvolvimento usado. Daí, o manifesto ágil proclama a importância das pessoas ao invés de processos. Se a equipe tem um elevado nível de capacidade e experiência, provavelmente a qualidade de produto será alta, independentemente do processo usado. Se a equipe for inexperiente e não qualificada, um bom processo pode limitar os danos, mas, por si mesmo, ele não vai gerar software de alta qualidade.

Figura 26.1 Fatores que afetam o produto de software

No caso de equipes pequenas, boas tecnologias de desenolvimento são particularmente importantes. Uma equipe pequena não pode dedicar muito tempo aos procedimentos administrativos tediosos. Os membros de equipe passam a maior parte do tempo em projeto e programação do sistema, portanto, boas ferramentas afetam significativamente sua produtividade. Quando se tratam de grandes projetos, um nível básico de tecnologia de desenvolvimento é essencial para o gerenciamento de informações. No entanto, paradoxalmente, as ferramentas sofisticadas de software são menos importantes em grandes projetos. Os membros de equipe gastam menos tempo em atividades de desenvolvimento e mais tempo se comunicando e compreendendo as outras partes do sistema. As ferramentas de desenvolvimento não fazem diferença nisso. No entanto, as ferramentas de Web 2.0 que suportam as comunicações, tais como *wikis* e blogs, podem melhorar significativamente as comunicações entre os membros de equipes distribuídas.

Independentemente de fatores, pessoas, processo ou ferramenta, se um projeto tiver um orçamento insuficiente ou se for planejado com um cronograma de entrega irrealista, a qualidade de produto será afetada. Um bom processo requer recursos para sua implementação efetiva. Se esses recursos não forem suficientes, o processo não poderá ser realmente eficaz. Se os recursos forem insuficientes, apenas pessoas excelentes poderão salvar um projeto. Mesmo assim, se o déficit for demasiado grande, a qualidade de produto ficará degradada. Se não houver tempo suficiente para o desenvolvimento, o software entregue pode ter funcionalidade reduzida ou níveis menores de desempenho ou confiabilidade.

Muitas vezes, a verdadeira causa dos problemas de qualidade de software não é o mau gerenciamento, processos inadequados ou treinamento ruim de qualidade. Na verdade, é o fato de que as organizações devem concorrer para sobreviver. Para obter um contrato, uma empresa pode subestimar o esforço necessário ou prometer a entrega rápida de um sistema. Na tentativa de atender a esses compromissos, elas podem concordar com um cronograma de desenvolvimento irrealista. Consequentemente, a qualidade do software será prejudicada.

26.1 O processo de melhoria de processos

No Capítulo 2, eu apresentei a ideia geral de um processo de software como uma sequência de atividades que, quando executadas, levam à produção de um sistema de software. Eu descrevi processos genéricos, tais como o modelo em cascata e o desenvolvimento baseado em reúso, e discuti as atividades mais importantes de processo. Esses processos genéricos são instanciados dentro de uma organização para criarem o processo particular, que a organização usa para desenvolver o software.

Os processos de software podem ser observados em todas as organizações, desde empresas com uma pessoa até as grandes multinacionais. Esses processos são de tipos diferentes, dependendo do grau de formalidade do processo, dos tipos de produtos desenvolvidos, do tamanho da organização e assim por diante. Não existe algo como um processo de software 'ideal' ou 'padrão' que seja aplicável a todas as organizações ou para todos os produtos de software de um tipo particular. Cada empresa precisa desenvolver seu próprio processo dependendo de seu tamanho, de seu *background* e das habilidades de seu pessoal, do tipo de software que esteja sendo desenvolvido, do cliente e dos requisitos de mercado, bem como da cultura da empresa.

Portanto, a melhoria dos processos não significa apenas a adoção de métodos particulares ou ferramentas ou o uso de um processo genérico publicado. Embora as organizações que desenvolvem o mesmo tipo de software, claramente, tenham muito em comum, sempre existem fatores organizacionais locais, procedimentos e normas que influenciam o processo. Raramente, você será bem-sucedido na introdução de melhorias

de processos, caso tente alterar o processo para um processo usado em outros lugares. Você sempre deve considerar o ambiente e a cultura locais e como esses fatores podem ser afetados por propostas de mudanças de processos.

Você também deve considerar quais aspectos do processo você deseja melhorar. Seu objetivo pode ser a melhoria da qualidade de software e, assim, você pode querer introduzir novas atividades de processo que alterem a forma como o software é desenvolvido e testado. Você pode estar interessado em melhorar algum atributo do processo em si e precisa decidir quais atributos de processo são mais importantes para sua empresa. Exemplos de atributos de processos que podem ser alvos de melhoria são mostrados na Tabela 26.1.

Esses atributos são relacionados, às vezes, positivamente; outras vezes, negativamente. Portanto, um processo com números altos no atributo visibilidade provavelmente também será compreensível. Um observador de processos pode inferir a existência de atividades, a partir das saídas produzidas. Por outro lado, a visibilidade de processo pode ser inversamente relacionada com a rapidez. Tornar um processo visível exige que as pessoas envolvidas produzam informações sobre o próprio processo. Isso pode diminuir a produção de software por causa do tempo necessário para produzir esses documentos.

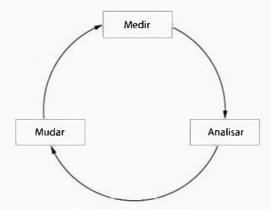
Não é possível fazer melhorias no processo que otimizem todos os atributos de processos simultaneamente. Por exemplo, se seu objetivo é ter um processo de desenvolvimento rápido, então você pode ter de reduzir a visibilidade de processo. Se você deseja fazer um processo mais manutenível, pode ser necessário adotar procedimentos e ferramentas que reflitam práticas organizacionais mais amplas, usadas em diferentes partes da empresa. Isso pode reduzir a aceitabilidade local do processo. Os engenheiros podem ter introduzido procedimentos locais e ferramentas não padronizadas para apoiarem sua forma de trabalhar. Como estas são eficazes, eles podem não querer desistir delas em favor de um processo padronizado.

O processo de melhoria de processos é um processo cíclico, conforme mostrado na Figura 26.2. Ele envolve três subprocessos:

1. *Medição de processo*. Os atributos do projeto atual ou dos produtos são medidos. O objetivo é melhorar as medidas de acordo com os objetivos da organização envolvida na melhoria de processos. Isso forma uma *baseline* que o ajuda a decidir se as melhorias no processo foram eficazes.

Tabela 26.1 Atributos de processo

Características de processo	Questões principais
Compreensibilidade	Em que medida o processo é definido explicitamente e quão fácil é entender sua definição?
Padronização	Em que medida o processo é baseado em um processo genérico padrão? Isso pode ser importante para alguns clientes que exigem conformidade com um conjunto definido de padrões de processos. Em que medida o mesmo processo é usado em todas as partes de uma empresa?
Visibilidade	As atividades do processo culminam em resultados claros, de modo que o progresso do processo seja visível externamente?
Capacidade de medição	O processo inclui a coleta de dados ou outras atividades que permitam que as características do produto ou processo sejam medidas?
Capacidade de apoio	Em que medida as ferramentas de software podem ser usadas para apoiar as atividades de processo?
Aceitabilidade	O processo definido é aceitável e usável pelos engenheiros responsáveis pela produção do produto de software?
Confiabilidade	O processo é projetado de forma que erros de processo sejam evitados ou encontrados antes de resultarem em erros de produto?
Robustez	O processo pode continuar apesar de problemas inesperados?
Manutenibilidade	O processo pode evoluir para refletir a mudança de requisitos organizacionais ou melhorias de processos identificados?
Rapidez	Quão rápido pode ser concluído o processo de entrega de um sistema após a conclusão de uma determinada especificação?

Figura 26.2 O ciclo de melhorias de processos

2. *Análise de processo.* O processo atual é avaliado e os gargalos e pontos fracos são identificados. Os modelos de processo (às vezes, chamados de mapas de processo) que descrevem o processo podem ser desenvolvidos durante essa fase. A análise pode centrar-se levando em consideração as características de processo, como rapidez e robustez.
3. *Mudanças de processo.* As mudanças de processos são propostas para resolver alguns dos pontos fracos identificados no processo. Elas são introduzidas e o ciclo recomeça para a coleta de dados sobre a eficácia das mudanças.

Sem dados concretos sobre um processo ou sobre o software desenvolvido que usa esse processo, é impossível avaliar o valor da melhoria de processos. No entanto, as empresas que iniciam o trabalho de melhoria de processos podem ter dados disponíveis como uma *baseline* de melhoria. Portanto, como parte do primeiro ciclo de mudanças, você pode ter de introduzir as atividades de processo para coletar dados sobre o processo de software e para medir as caraterísticas de produto de software.

A melhoria de processos é uma atividade de longo prazo, pois cada uma das fases do processo de melhoria pode durar vários meses. Também é uma atividade contínua, pois quaisquer novos processos introduzidos alterarão o ambiente de negócios e os novos processos terão de evoluir para levar em conta essas mudanças.

26.2 Medição de processos

As medições de processos são dados quantitativos sobre o processo de software, tal como o tempo necessário para realizar alguma atividade do processo. Por exemplo, você pode medir o tempo necessário para o desenvolvimento de casos de teste de programa. Humphrey (1989), em seu livro sobre a melhoria de processos, argumenta que a medição de atributos de processo e produto é essencial para a melhoria de processos. Ele também sugere que a medição tem um papel importante na melhoria de processos pessoais de pequena escala (HUMPHREY, 1995), em que os indivíduos tentam tornar-se mais produtivos.

As medições de processo podem ser usadas para avaliar a melhoria da eficiência de um processo. Por exemplo, o esforço e o tempo dedicado aos testes podem ser monitorados. Melhorias eficazes no processo de teste devem reduzir o esforço e/ou tempo de teste. No entanto, as medições de processo por si não podem ser usadas para determinar se a qualidade de produto melhorou. Os dados de qualidade de produto (ver Capítulo 24) também devem ser coletados e relacionados com as atividades de processo.

Três tipos de métricas de processo podem ser coletados:

1. *O tempo necessário para um processo específico ser concluído.* Esse pode ser o tempo total dedicado para o processo, o tempo de calendário, o tempo dispendido no processo por engenheiros particulares, e assim por diante.
2. *Os recursos necessários para um determinado processo.* Os recursos podem incluir total empenho em pessoas/dia, custos de viagens ou recursos de computador.
3. *O número de ocorrências de um determinado evento.* Exemplos de eventos que podem ser monitorados incluem o número de defeitos descobertos durante a inspeção de código, o número de mudanças de requisitos solicitados e o número médio de linhas de código modificadas em resposta a uma mudança de requisitos.

Os dois primeiros tipos de medição podem ser usados para descobrir se as mudanças no processo melhoraram a eficiência de um processo. Digamos que haja pontos fixos em um processo de desenvolvimento de software, tais como a aceitação de requisitos, a conclusão do projeto de arquitetura ou a conclusão da geração de dados de teste. Você pode ser capaz de medir o tempo e o esforço necessários para mover de um desses pontos para outro ponto fixo. Depois que as mudanças foram introduzidas, as medições de atributos de sistema podem mostrar se as mudanças de processo foram bem-sucedidas em reduzir o tempo ou o esforço requerido.

As medições do número de eventos ocorridos têm uma relação mais direta sobre a qualidade de software. Por exemplo, o aumento do número de defeitos descobertos pela mudança do processo de inspeção de programa provavelmente se refletirá na melhoria da qualidade de produto. No entanto, isso deve ser confirmado por meio de medições de produto subsequentes.

Uma dificuldade fundamental no processo de medição é saber quais informações sobre o processo devem ser coletadas para apoiar a melhoria de processo. Basili e Rombach (1988) propuseram o que eles chamam de paradigma GQM (Metas-questões-métricas, do inglês *Goal-Question-Metric*), que tem se tornado amplamente usado na medição de softwares e processos. Basili e Green (1993) descrevem como essa abordagem foi usada em um programa de medição de longo prazo para a melhoria de processos, na Agência Espacial dos Estados Unidos NASA.

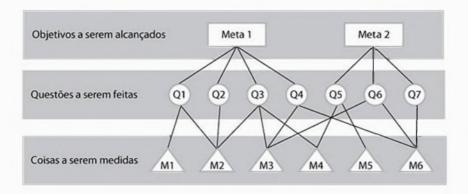
O paradigma GQM (Figura 26.3) é usado na melhoria de processos para ajudar a responder três questões críticas:

1. Por que estamos introduzindo melhorias de processos?
2. De quais informações precisamos para ajudar a identificar e avaliar as melhorias?
3. Quais medições de processo e produto são necessárias para fornecer essas informações?

Essas questões estão diretamente relacionadas com as abstrações (metas, questões, métricas) do paradigma GQM:

1. *Metas*. Uma meta é algo que a organização está tentando atingir. Ela não deve ser diretamente relacionada aos atributos de processo, mas sim a como o processo afeta os produtos ou a própria organização. Exemplos de metas podem ser um melhor nível de maturidade de processo (veja a Seção 26.5), menor tempo de desenvolvimento de produto ou aumento de confiabilidade de produto.
2. *Questões*. Estas são refinamentos das metas, nas quais foram identificadas áreas específicas de incertezas relacionadas às metas. Normalmente, uma meta terá várias questões associadas, as quais precisam ser respondidas. Exemplos de questões relacionadas à meta de reduzir o tempo de desenvolvimento de produto podem ser 'Onde estão os gargalos de nosso processo atual?', 'Como podemos reduzir o tempo necessário para finalizar os requisitos de produto?' e 'Quantos dos nossos testes são eficazes na descoberta de defeitos de produtos?'.
3. *Métricas*. Estas são as medidas que precisam ser coletadas para ajudar a responder as questões e para confirmar se as melhorias de processos alcançaram a meta desejada. Para ajudar a responder as questões anteriores, você pode coletar dados sobre o tempo levado para concluir cada atividade do processo (normalizado pelo tamanho de sistema), o número de comunicações formais entre os clientes e desenvolvedores para cada mudança de requisitos e o número de defeitos descobertos por meio de execução de testes.

A vantagem de usar a abordagem GQM na melhoria de processos é que ela separa os interesses organizacionais (as metas) dos interesses específicos de processos (as questões). Ela fornece uma base para decidir quais dados devem ser coletados e sugere que os dados coletados devem ser analisados de diferentes maneiras, dependendo da questão que devem responder.

Figura 26.3 O paradigma GQM

A abordagem GQM foi desenvolvida e combinada com o modelo de maturidade e de capacidade do SEI (PAULK et al., 1995) no método AMI (Analisar, Medir, Melhorar, do inglês *Analyze, Measure, Improve*) de melhoria de processos de software. Os desenvolvedores do método AMI propõem uma abordagem por estágios para processo de melhoria, em que a medição é iniciada após uma organização introduzir alguma padronização em seus processos, ao contrário da medição de início imediato. O manual de AMI (PULFORD et al., 1996) fornece diretrizes e conselhos práticos sobre a implementação de melhoria de processos baseados em medições.

Como discutido no Capítulo 24, interpretar o que realmente significam as medições pode ser problemático. Por exemplo, digamos que você mede o tempo médio para reparar *bugs* relatados nos softwares que foram entregues para testes externos. Esse é o tempo entre a equipe receber um relatório de erros e o tempo em que esse relatório é marcado como 'limpo'. Você introduz uma nova ferramenta baseada na Web para relatórios de erros e, depois dessa ferramenta ter sido usada por algum tempo, você observa se o tempo para reparar os *bugs* relatados foi reduzido.

Em seguida, é possível afirmar que a introdução de ferramentas de relatórios de erros reduziu o tempo para reparar os *bugs*. Quando você observa mudanças em uma métrica, é tentador atribuir essas mudanças para as mudanças de processo que você introduziu. No entanto, é perigoso fazer suposições simplistas sobre as melhorias. As mudanças em uma métrica podem ser causadas por algo completamente diferente, tais como uma mudança de pessoas na equipe de projeto, mudanças no cronograma de projeto ou mudanças de gerenciamento. Também pode acontecer que as práticas da equipe mudem simplesmente por estarem sendo medidas. No caso de ferramentas de geração de relatórios de erros, alguns dos motivos pelos quais se observou a mudança incluem:

1. O novo sistema pode ter reduzido o *overhead*, de modo que há mais tempo disponível para reparação de *bugs*. Isso leva a uma redução nos tempos médios de 'reparação de *bugs*'. A melhoria de processo pode ter feito uma diferença verdadeira.
2. O novo sistema pode não ter feito diferença alguma no tempo real necessário para corrigir *bugs*, mas ele pode ter facilitado o registro de informações. Portanto, o tempo de reparação de *bugs* é mensurado mais precisamente com o novo sistema. Não houve mudança real no tempo médio de correção de *bugs*.
3. As medições antes da introdução do novo sistema foram, talvez, feitas em parte por meio de testes de um sistema. Os *bugs* que foram mais fáceis e mais rápidos de serem corrigidos já tinham sido corrigidos e apenas os '*bugs* mais difíceis' permaneceram, os quais tomam mais tempo para serem reparados. No entanto, após o sistema ser introduzido, medições são feitas no início dos testes do novo sistema e os *bugs* corrigidos são os '*bugs* fáceis' que poderiam ser reparados rapidamente.
4. Um novo gerente da equipe de testes pode ter instruído os membros de equipe a relatarem as inconsistências de interface do usuário como *bugs*, considerando que antes elas foram ignoradas. Isso significa que *bugs* 'mais fáceis' foram relatados, os quais poderiam ser resolvidos rapidamente.

A medição é uma forma de gerar evidências sobre um processo e mudanças de processo. No entanto, essas evidências precisam ser interpretadas juntamente com outras informações sobre o processo antes que você possa ter certeza de que as mudanças de processo são eficazes. Você sempre deve usar a medição em conjunto com a avaliação qualitativa das mudanças. Isso envolve conversar com as pessoas envolvidas no processo a respeito das mudanças introduzidas e receber suas impressões sobre a eficácia dessas mudanças. Esse processo revela outros fatores que podem ter influenciado o processo, como também revela a extensão com que a equipe adotou as mudanças propostas e como elas têm afetado a prática real de desenvolvimento.

26.3 Análise de processos

A análise de processo é o estudo dos processos para ajudar a compreender suas principais características e como esses processos são executados na prática, pelas pessoas envolvidas. Na Figura 26.2, eu sugiro que a análise de processos segue a medição de processo. Isso é uma simplificação, pois, na realidade, essas atividades estão intercaladas. É preciso realizar algumas análises para saber o que medir, e, quando estiver fazendo as medições, você inevitavelmente desenvolverá uma compreensão mais profunda do processo a ser medido.

A análise de processos tem vários objetivos estreitamente relacionados:

1. Entender as atividades envolvidas no processo e os relacionamentos entre essas atividades.
2. Entender os relacionamentos entre as atividades de processo e as medições feitas.

3. Relacionar o processo específico ou processos que você está analisando com processos comparáveis localizados em outro lugar na organização, ou com processos idealizados de mesmo tipo.

Durante a análise de processos, você está tentando entender o que está acontecendo em um processo. Você procura informações sobre os problemas e as ineficiências do processo; você também deve estar interessado em que medida o processo é usado, nas ferramentas de software usadas para apoiar o processo e em como o processo é influenciado por restrições organizacionais. A Tabela 26.2 mostra alguns dos aspectos do processo que você pode investigar durante a análise de processos.

As técnicas mais usadas na análise de processos são:

1. *Questionários e entrevistas*. Os engenheiros e gerentes que trabalham em um projeto são questionados sobre o que realmente está acontecendo. As respostas ao questionário formal são refinadas durante entrevistas pessoais com os envolvidos no processo. Como vemos adiante, a discussão pode ser estruturada em torno de modelos de processos de software.
2. *Estudos etnográficos*. Estudos etnográficos (veja o Capítulo 4), em que os participantes de processo são observados enquanto trabalham, podem ser usados para compreender a natureza do desenvolvimento de software como uma atividade humana. Essa análise revela sutilezas e complexidades que não podem ser reveladas por questionários e entrevistas.

Cada uma dessas abordagens tem vantagens e desvantagens. A análise baseada em questionários pode ser realizada rapidamente uma vez que tenham sido identificadas as questões certas. No entanto, se as questões forem inapropriadas ou mal redigidas, pode-se ter uma compreensão incompleta ou imprecisa do processo. Além disso, a análise baseada em questionários pode parecer uma forma de avaliação. Portanto, os engenheiros questionados podem responder o que eles acreditam que você gostaria de ouvir, em vez de a verdade sobre o processo usado.

Tabela 26.2 Aspectos da análise de processos

Aspecto de processo	Questões
Adoção e padronização	O processo está documentado e padronizado em toda a organização? Se não, isso significa que quaisquer medições efetuadas são específicas apenas para uma única instância de processo? Se os processos não forem padronizados, então as mudanças para um processo podem não ser transferíveis para processos comparáveis, em outros lugares da empresa.
Prática de engenharia de software	Existem boas práticas de engenharia de software, que não estão incluídas no processo? Por que elas não estão incluídas? A falta dessas práticas afeta as características de produto, tal como o número de defeitos em um sistema de software entregue?
Restrições organizacionais	Quais são as restrições organizacionais que afetam o projeto de processo e as maneiras como o processo é executado? Por exemplo, se o processo envolve lidar com material confidencial, pode haver atividades no processo para verificar se as informações confidenciais não estão incluídas em qualquer material que será liberado para organizações externas. Restrições organizacionais podem significar que possíveis mudanças de processo não podem ser realizadas.
Comunicações	Como as comunicações são gerenciadas no processo? Como os problemas de comunicação se relacionam com as medições de processo feitas? Os problemas de comunicação são uma questão importante em muitos processos e, muitas vezes, os gargalos de comunicação são motivo para atrasos no projeto.
Introspecção	O processo é reflexivo (ou seja, os atores envolvidos no processo pensam e discutem o processo e como ele pode ser melhorado explicitamente)? Existem mecanismos pelos quais os atores de processo podem propor melhorias de processo?
Aprendizagem	Como as pessoas que integram uma equipe de desenvolvimento aprendem sobre os processos de software usados? A empresa tem manuais dos processos e programas de treinamento de processos?
Suporte a ferramentas	Quais aspectos do processo são e não são suportados por ferramentas de software? Para áreas sem suporte, existem ferramentas que poderiam ser implantadas efetivamente, para fornecer suporte? Para áreas com suporte, as ferramentas são eficazes e eficientes? As melhores ferramentas estão disponíveis?

Entrevistas com as pessoas envolvidas no processo são menos limitadas do que questionários. Você começa com um *script* preparado de questões, mas adapta-as de acordo com as respostas obtidas. Se você der aos participantes uma oportunidade de discutir as questões mais amplamente, você pode achar que os participantes do processo falam sobre problemas de processo, as maneiras como o processo é modificado na prática e assim por diante.

Em quase todos os processos, as pessoas envolvidas fazem mudanças locais para adaptar os processos de acordo com as circunstâncias locais. É mais provável que a análise etnográfica descubra o verdadeiro processo usado, e não as entrevistas. No entanto, esse tipo de análise pode ser uma atividade prolongada, que pode durar vários meses. Ela se baseia na observação externa do processo, à medida que está sendo aprovado. Para fazer uma análise completa, você precisa estar envolvido desde os estágios iniciais de um projeto, passando pela entrega de produto até a manutenção. Para grandes projetos, isso pode levar vários anos, por isso é claramente impraticável fazer uma análise etnográfica completa dos processos em um projeto de grande porte. A análise etnográfica é realmente útil quando é necessário um entendimento profundo dos fragmentos de processos. Depois de identificar áreas que demandam outras investigações do material das entrevistas, você pode fazer um estudo etnográfico centrado em descobrir detalhes de processo.

Ao se analisar um processo, costuma ser útil iniciar com um modelo de processo que defina as atividades no processo e as entradas e saídas dessas atividades. O modelo também pode incluir informações sobre os atores de processo — as pessoas ou os papéis responsáveis pela execução das atividades e os resultados essenciais que devem ser apresentados. Você pode usar uma notação informal para descrever modelos de processo ou notações tabulares mais formais, diagramas de atividade de UML ou uma notação de modelagem de processo de negócios, como BPMN (discutido no Capítulo 19). Existem muitos exemplos de modelos de processo neste livro, os quais eu uso para apresentar e descrever processos de software.

Os modelos de processo são uma boa maneira de centrar a atenção nas atividades em um processo e na transferência de informações entre essas atividades. Esses modelos de processo não precisam ser formais ou completos — seu objetivo é provocar discussão em vez de documentar o processo detalhadamente. Muitas vezes, a discussão com as pessoas envolvidas no processo e as observações desse processo são estruturadas em torno de um conjunto de questões sobre o modelo de processo formal. Exemplos dessas questões podem ser:

1. Quais atividades ocorrem na prática, mas não são mostradas no modelo? Inevitavelmente, os modelos estão incompletos, mas se pessoas diferentes identificam a ausência de diferentes atividades, isso significa que o processo não está sendo executado consistentemente em toda a organização.
2. Existem atividades de processos, mostradas no modelo, que você (o ator de processo) acredita serem ineficientes? De que forma elas são ineficientes e como podem ser melhoradas? Como essas atividades ineficientes afetam as medições de processo que podem ter sido feitas?
3. O que acontece quando as coisas dão errado? A equipe continua a seguir o processo definido no modelo ou o processo é abandonado e são tomadas medidas de emergência? Se o processo for abandonado, isso sugere que os engenheiros de software não acreditam que o processo seja bom o suficiente ou que ele não tem suficiente flexibilidade para tratar exceções.
4. Quem são os atores envolvidos nos diferentes estágios do processo e como eles se comunicam? Quais gargalos geralmente ocorrem na troca de informações?
5. Qual suporte de ferramentas é usado para as atividades mostradas no modelo? Isso é eficaz e universalmente usado? Como o suporte de ferramentas poderia ser melhorado?

Ao concluir uma análise do processo de software, você deve ter uma compreensão mais profunda desse processo e do potencial de melhoria de processos no futuro. Você também deve compreender as restrições nas melhorias de processos e como estas podem limitar o escopo de melhorias que podem ser introduzidas.

26.3.1 Exceções de processos

Os processos de software são entidades complexas. Pode haver um modelo de processo definido em uma organização, mas ele só pode representar a situação em que a equipe de desenvolvimento não enfrenta quaisquer problemas imprevistos. Na realidade, problemas inesperados são um fato no cotidiano dos gerentes de projeto. O modelo de processo 'ideal' deve ser dinamicamente modificado, conforme se encontrem soluções para esses problemas. Alguns exemplos de tipos de exceção com que um gerente de projeto pode ter de lidar são:

- várias pessoas importantes adoecerem ao mesmo tempo, pouco antes de uma revisão crítica de projeto;
- uma violação grave na proteção de um computador, significando que todas as comunicações externas estarão inativas por vários dias;
- uma reorganização de empresa, significando que os gerentes gastam parte significativa de seu tempo de trabalho com questões organizacionais, em vez de gerenciamento de projetos;
- uma solicitação imprevista para escrever uma proposta de um novo projeto, significando que o esforço deve ser transferido do projeto atual para a escrita da proposta.

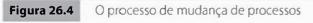
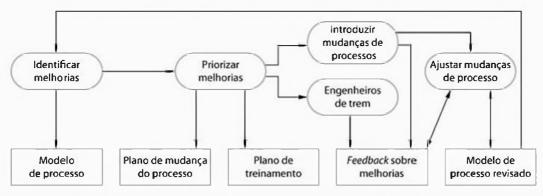
Essencialmente, uma exceção afetará e, geralmente, alterará de alguma forma os recursos, orçamentos ou cronogramas de um projeto. É difícil prever todas as exceções antecipadamente e incorporá-las em um modelo de processo formal. Por isso, muitas vezes você precisa definir como tratar as exceções e, em seguida, mudar dinamicamente o processo-'padrão' para lidar com essas circunstâncias inesperadas.

26.4 Mudança de processos

As mudanças de processo envolvem fazer modificações no processo existente. Como sugeri, você pode fazer isso por meio da introdução de novas práticas, métodos ou ferramentas; mudando a ordem das atividades do processo; introduzindo ou removendo os entregáveis do processo; melhorando as comunicações; ou introduzindo novos papéis e responsabilidades. As mudanças de processo devem ser conduzidas por metas de melhoria, tais como 'reduzir o número de defeitos descobertos durante os testes de integração em 25%'. Depois de implementadas as mudanças, você usa as medições de processo para avaliar a eficácia das mudanças.

Existem cinco estágios principais no processo de mudança de processos (Figura 26.4):

1. *Identificação de melhorias*. Esse estágio está relacionado com o uso dos resultados da análise de processo para identificar maneiras de lidar com os problemas de qualidade, gargalos de cronograma ou ineficiências de custo identificadas durante a análise de processo. Você pode propor novos processos, estruturas de processo, métodos e ferramentas para resolver problemas de processo. Por exemplo, uma empresa pode acreditar que muitos de seus problemas de software resultam de problemas de requisitos. Usando um guia de melhores práticas (SOMMERVILLE e SAWYER, 1997) de engenharia de requisitos, podem ser identificadas várias práticas de engenharira de requisitos que podem ser introduzidas ou alteradas.
2. *Priorização de melhorias*. Esse estágio está relacionado com avaliação de possíveis mudanças no processo e em priorização dessas mudanças para sua implementação. Quando se identificam muitas possíveis mudanças, é normalmente impossível apresentar todas ao mesmo tempo e você deve decidir quais são as mais importantes. Você pode tomar essas decisões com base na necessidade de melhorar áreas de processo específicas, nos custos de se introduzir uma mudança, no impacto de uma mudança na organização, ou em outros fatores. Por exemplo, uma empresa pode considerar a introdução de processos de gerenciamento de requisitos para gerenciar requisitos em evolução para ser a mudança de processo com prioridade mais alta.
3. *Introdução de mudanças de processo*. A introdução de mudanças de processo significa colocar novos procedimentos, métodos e ferramentas no lugar e integrá-los com outras atividades de processo. Você deve permi-

Figura 26.4 O processo de mudança de processos

tir tempo suficiente para a introdução de mudanças e garantir que essas mudanças sejam compatíveis com outras atividades de processos, bem como procedimentos e padrões organizacionais. Isso pode envolver a aquisição de ferramentas para gerenciamento de requisitos e o projeto de processos para usar tais ferramentas.

4. *Treinamento de processos*. Sem treinamento, não é possível obter todos os benefícios das mudanças de processo. Os engenheiros envolvidos precisam entender as mudanças que foram propostas e como executar os processos novos e alterados. Muitas vezes, as mudanças de processo são impostas sem o treinamento adequado e o efeito dessas mudanças é a degradação, em vez da melhoria, da qualidade de produto. No caso do gerenciamento de requisitos, o treinamento pode envolver uma discussão sobre o valor do gerenciamento de requisitos, uma explicação sobre as atividades de processo e uma introdução às ferramentas selecionadas.
5. *Ajuste de mudanças*. As mudanças de processos propostas nunca serão completamente eficazes assim que são introduzidas. Você precisa de uma fase de ajustes, na qual problemas menores podem ser descobertos e modificações para o processo podem ser propostas e introduzidas. Essa fase de ajuste deve durar vários meses até que os engenheiros de desenvolvimento estejam satisfeitos com o novo processo.

Geralmente, é imprudente introduzir muitas mudanças ao mesmo tempo. Além das dificuldades de treinamento, a introdução de muitas mudanças impossibilita a avaliação do efeito de cada uma no processo. Uma vez que uma mudança tenha sido introduzida, o processo de melhoria pode iterar e novas análises podem ser usadas para identificar problemas de processo, propor melhorias e assim por diante.

Assim como as dificuldades de avaliação da eficácia dos processos alterados, existem duas grandes dificuldades que os envolvidos nos processos de mudança podem enfrentar:

1. *Resistência às mudanças*. Os membros de equipe ou gerentes de projeto podem resistir à introdução de mudanças de processo e sugerir razões pelas quais as mudanças não funcionarão, ou também podem atrasar a introdução delas. Em alguns casos, eles podem obstruir as mudanças de processo deliberadamente e interpretar dados para mostrar a ineficácia das mudanças do processo proposto.
2. *Persistência de mudanças*. Embora possa ser possível introduzir mudanças de processo, inicialmente, é comum que as inovações de processo sejam descartadas após um curto período e que os processos retornem a seu estado anterior.

A resistência às mudanças pode vir tanto dos gerentes de projeto quanto dos engenheiros envolvidos no processo que está sendo alterado. Muitas vezes, os gerentes de projeto resistem às mudanças de processo, pois quaisquer inovações carregam riscos desconhecidos. As mudanças de processo podem ser destinadas a acelerar a produção de softwares ou reduzir defeitos neles. No entanto, sempre existe o perigo de que essas mudanças de processo sejam ineficientes ou que o tempo necessário para introduzir as mudanças seja maior do que o tempo economizado. Os gerentes de projeto são avaliados de acordo com a capacidade do projeto de produzir software dentro do prazo e orçamento. Portanto, podem preferir um processo ineficiente, mas previsível, a um processo melhorado que tenha benefícios organizacionais, mas que, no curto prazo, possa ter riscos associados a ele.

Os engenheiros podem resistir à introdução de novos processos por entenderem tais processos como, dentre outras razões, uma ameaça a seu profissionalismo. Ou seja, eles podem sentir que o novo processo predefinido lhes dá menos liberdade de ação e não reconhece o valor de suas habilidades e experiência. Podem pensar que o novo processo significa que menos pessoas serão necessárias e que eles podem perder o emprego. Além disso, eles podem não querer aprender novas habilidades, ferramentas ou métodos de trabalho.

Como gerente, você precisa ser sensível aos sentimentos das pessoas afetadas pela introdução de mudanças de processo. Você precisa envolver a equipe no processo de mudança, compreender suas dúvidas e envolvê-la no planejamento do novo processo. Ao tornar sua equipe *stakeholders* da mudança de processo, é muito mais provável que eles queiram fazer o processo funcionar. A reengenharia de processo de negócios (HAMMER, 1990; OULD, 1995), uma moda da década de 1990 que envolveu mudanças radicais de processo, fracassou em grande parte porque não conseguiu levar em consideração as preocupações das pessoas envolvidas.

Para lidar com os interesses dos gerentes de projeto cujo processo de mudança afetará negativamente os custos e cronogramas de projeto, você precisa aumentar os orçamentos de projeto a fim de permitir os custos adicionais e atrasos resultantes da mudança. Você também precisa ser realista sobre os benefícios a curto prazo da mudança. As mudanças podem conduzir a melhorias imediatas, de grande escala. Os benefícios da mudança de processos são de longa duração e não de curto prazo, portanto, é necessário apoiar as mudanças de processos em vários projetos.

O problema de mudanças serem introduzidas e posteriormente descartadas é comum. As mudanças podem ser propostas por um 'evangelista' que acredita firmemente que elas levarão a melhorias. Ele pode trabalhar ar-

duamente para garantir que as mudanças sejam eficazes e que o novo processo seja aceito. No entanto, se esse 'evangelista' vai embora, ele pode ser substituído por alguém menos comprometido com o novo processo. Portanto, as pessoas envolvidas podem voltar a fazer as coisas como antes. Isso é bastante provável, caso as mudanças introduzidas não tenham sido universalmente adotadas e os benefícios completos das mudanças de processo não tenham sido alcançados.

Em virtude dos problemas de persistência de mudanças, o modelo CMMI, discutido na Seção 26.5, defende fortemente a institucionalização da mudança de processo. Isso significa que o processo de mudança não é dependente de indivíduos, mas que as mudanças se tornam parte da prática-padrão da empresa, com treinamentos e o apoio de toda a empresa.

26.5 *Framework* CMMI de melhorias de processos

O Instituto de Engenharia de Software (SEI, do inglês *Software Engineering Institute*) dos Estados Unidos foi criado visando a melhoria das capacidades da indústria norte-americana de software. Em meados de 1980, o SEI iniciou um estudo sobre como avaliar as capacidades dos prestadores de serviços de software. O resultado dessa avaliação de capacidade foi o Modelo Maturidade e de Capacidade (CMM, do inglês *Capability Maturity Model*) do SEI (PAULK et al., 1993; PAULK et al., 1995). Este tem sido tremendamente influente em convencer a comunidade de engenharia de software a levar a sério a melhoria de processos. O CMM para software foi seguido por uma gama de outros modelos de maturidade e de capacidade, incluindo o Modelo de Maturidade e de Capacidade de Pessoas (CMM-P, do inglês *People Capability Maturity Model*) (CURTIS et al., 2001) e o Modelo de Capacidade de Engenharia de Sistemas (BATE, 1995).

Outras organizações também desenvolveram modelos de maturidade de processos comparáveis. A abordagem SPICE para avaliação de capacidade e melhoria de processos (PAULK e KONRAD, 1994) é mais flexível do que o modelo do SEI. Ela inclui níveis de maturidade comparáves aos níveis do CMM, mas também identifica processos, tais como os processos de cliente-fornecedor, que atravessam esses níveis. À medida que o nível de maturidade aumenta, o desempenho desses processos transversais também deve melhorar.

O projeto Bootstrap, na década de 1990, tinha o objetivo de estender e adaptar o modelo de maturidade do SEI para torná-lo aplicável a uma vasta gama de empresas. Esse modelo (HAASE et al., 1994; KUVAJA et al., 1994) usa níveis de maturidade do SEI (discutidos na Seção 26.5.1). Ele também propõe um modelo de processo-base (baseado no modelo usado na Agência Espacial Europeia) que pode ser usado como ponto de partida para a definição de processo local. Além de incluir diretrizes para o desenvolvimento de um sistema de qualidade de toda a empresa para apoiar a melhoria de processos.

Em uma tentativa de integrar a multiplicidade de modelos de capacidade com base na noção de maturidade de processos (incluindo seus próprios modelos), o SEI embarcou em um novo programa para desenvolver um modelo de capacidade integrado (CMMI). O *framework* do CMMI substitui os CMMs para Software e Engenharia de Sistemas e integra outros modelos de maturidade e de capacidade. Ele tem duas instâncias, por estágio e contínua, e aborda alguns dos pontos fracos relatados no CMM para Software.

O modelo CMMI (AHERN et al., 2001; CHRISSIS et al., 2007) destina-se a ser um *framework* de melhoria de processos com ampla aplicabilidade em uma gama de empresas. Sua versão por estágios é compatível com o CMM para Software e permite que o desenvolvimento de sistemas e processos de gerenciamento de uma organização seja avaliado e que a ele seja atribuído um nível de maturidade classificado como de 1 a 5. Sua versão contínua permite uma classificação de granularidade mais baixa de maturidade de processo. Esse modelo fornece uma maneira de classificar 22 áreas de processo (veja a Tabela 26.3) em uma escala de 0 a 5.

O modelo CMMI é muito complexo, com mais de mil páginas de descrição. Para nossa discussão, eu simplifiquei radicalmente esse modelo. Os principais componentes do modelo são:

1. Um conjunto de áreas de processo relacionadas às atividades de processos de software. O CMMI identifica 22 áreas de processo relevantes para a melhoria e a capacidade de processo de software. Estas são organizadas em quatro grupos no modelo CMMI contínuo. Esses grupos e áreas de processo relacionadas são listados na Tabela 26.3.
2. Um número de metas, que são descrições abstratas de um estado desejável a ser atingido por uma organização. O CMMI tem metas específicas, associadas a cada área de processo, e define o estado desejável para cada área. Ele também define metas genéricas associadas com a institucionalização das boas práticas. A Tabela 26.4 mostra exemplos de metas específicas e genéricas no CMMI.

Tabela 26.3 Áreas de processo no CMMI

Categoria	Área de processo
Gerenciamento de processos	Definição de processo organizacional (OPD, do inglês *organizational process definition*) Foco de processo organizacional (OPF, do inglês *organizational process focus*) Treinamento organizacional (OT, do inglês *organizational training*) Desempenho de processo organizacional (OPP, do inglês *organizational process performance*) Inovação e implantação organizacional (OID, do inglês *organizational innovationn and deployment*)
Gerenciamento de projetos	Planejamento de projeto (PP, do inglês *project planning*) Monitoração e controle de projeto (PMC, do inglês *project monitoring and control*) Gerenciamento de acordo com fornecedores (SAM, do inglês *supplier agreement management*) Gerenciamento de projeto integrado (IPM, do inglês *integrated project management*) Gerenciamento de riscos (RSKM, do inglês *risk management*) Gerenciamento quantitativo de projeto (QPM, do inglês *quantitative project management*)
Engenharia	Gerenciamento de requisitos (REQM, do inglês *requirements management*) Desenvolvimento de requisitos (RD, do inglês *requirements development*) Solução técnica (TS, do inglês *technical solution*) Integração de produto (PI, do inglês *product integration*) Verificação (VER, do inglês *verification*) Validação (VAL, do inglês *validation*)
Suporte	Gerenciamento de configuração (CM, do inglês *configuration management*) Garantia de qualidade de processo e produto (PPQA, do inglês *process and product quality management*) Medição e análise (MA, do inglês *measurement and analysis*) Análise de decisão e resolução (DAR, do inglês *decision analysis and resolution*) Análise causal e resolução (CAR, do inglês *causal analysis and resolution*)

Tabela 26.4 Áreas de processo no CMMI

Meta	Área de processo
Ações corretivas são gerenciadas até a conclusão, quando o desempenho ou os resultados do projeto se desviam significativamente do plano.	Monitoração e controle de projeto (meta específica)
O desempenho real e o progresso do projeto são monitorados contra o plano de projeto.	Monitoração e controle de projeto (meta específica)
Os requisitos são analisados e validados, e uma definição da funcionalidade requerida é desenvolvida.	Desenvolvimento de requisitos (meta específica)
Causas-raiz de defeitos e outros problemas são sistematicamente determinados.	Análise causal e resolução (meta específica)
O processo é institucionalizado como um processo definido.	Meta genérica

3. Um conjunto de boas práticas, que são as descrições das formas de como alcançar uma meta. Várias práticas específicas e genéricas podem ser associadas com cada meta dentro de uma área de processo. Alguns exemplos de práticas recomendadas são mostrados na Tabela 26.5. No entanto, o CMMI reconhece que o mais importante é a meta e não a maneira como ela é alcançada. As organizações podem usar quaisquer práticas adequadas para atingir qualquer uma das metas do CMMI — elas não precisam adotar as práticas recomendadas no CMMI.

Metas e práticas genéricas não são técnicas, mas estão associadas com a institucionalização de boas práticas. O que isso significa depende da maturidade da organização. Em um estágio inicial de desenvolvimento de maturidade, a institucionalização pode significar a garantia de que os planos sejam estabelecidos e os processos, definidos para todo o desenvolvimento de software da empresa. No entanto, para uma organização com processos mais maduros e avançados, a institucionalização pode significar a introdução de controle de processo usando técnicas estatísticas e outras técnicas quantitativas em toda a organização.

Tabela 26.5 Metas e práticas associadas no CMMI

Meta	Práticas associadas
Os requisitos são analisados e validados, e uma definição da funcionalidade requerida é desenvolvida.	Analisar sistematicamente os requisitos derivados para garantir que eles são necessários e suficientes.
	Validar os requisitos para garantir que o produto resultante executará conforme pretendido no ambiente do usuário, usando várias técnicas conforme apropriado.
Causas-raiz de defeitos e outros problemas são sistematicamente determinados.	Selecionar os defeitos críticos e outros problemas para análise.
	Realizar uma análise causal de defeitos selecionados e outros problemas e propor ações para solucioná-los.
O processo é institucionalizado como um processo definido.	Estabelecer e manter uma política organizacional para planejar e executar o processo de desenvolvimento de requisitos.
	Atribuir responsabilidade e autoridade para executar o processo, desenvolver os produtos de trabalho e prestar os serviços do processo de desenvolvimento de requisitos.

Uma avaliação de CMMI envolve examinar os processos em uma organização e classificar esses processos ou áreas de processo em uma escala de seis pontos relacionada ao nível de maturidade em cada área de processo. A ideia é que quanto mais maduro for um processo, melhor. A escala de seis pontos atribui níveis de maturidade para uma área de processo, da seguinte forma:

1. *Incompleto*. Pelo menos uma das metas específicas associadas com a área de processo não está satisfeita. Não existem metas genéricas nesse nível, pois a institucionalização de um processo incompleto não faz sentido.
2. *Executado*. As metas associadas com a área de processo são satisfeitas e, para todos os processos, o escopo de trabalho a ser realizado é explicitamente definido e comunicado para os membros da equipe.
3. *Gerenciado*. Nesse nível, as metas associadas com a área de processo são cumpridas e as políticas organizacionais definem quando cada processo deve ser usado. Deve haver planos de projeto documentados, que definam as metas de projeto. Os procedimentos de gerenciamento de recursos e de controle de processos devem estar em ordem, em toda a instituição.
4. *Definido*. Esse nível concentra-se na padronização e implantação de processos organizacionais. Cada projeto tem um processo gerenciado que é adaptado aos requisitos de projeto de um conjunto definido de processos organizacionais. Ativos de processo e medições de processo devem ser coletados e usados para melhoria de processos.
5. *Gerenciado quantitativamente*. Nesse nível, existe uma responsabilidade organizacional de usar métodos estatísticos e outros métodos quantitativos para controlar os subprocessos; ou seja, as medições de processo e produto coletados devem ser usadas no gerenciamento de processos.
6. *Otimizando*. Nesse nível mais alto, a organização deve usar as medições de processo e produto para dirigir a melhoria de processos. Devem ser analisadas as tendências e os processos adaptados às mudanças de necessidades de negócios.

Essa é uma descrição muito simplificada dos níveis de capacidade e, para colocá-las em prática, você precisa trabalhar com descrições mais detalhadas. Os níveis são progressivos, com descrições de processos explícitas nos níveis mais baixos, por meio da padronização de processos, para mudança e melhoria de processos dirigidas por medições do processo e do software no nível mais alto. Para melhorar seus processos, uma empresa deve procurar aumentar o nível de maturidade dos grupos de processo relevantes para seu negócio.

26.5.1 O modelo CMMI por estágios

O modelo CMMI por estágios é comparável ao Modelo de Maturidade e de Capacidade de Software, que fornece um meio para avaliar a capacidade de processo de uma organização em um dos cinco níveis e prescreve as metas que devem ser alcançadas em cada um desses níveis. A melhoria de processos é alcançada por meio da implementação de práticas em cada nível, desde os níveis inferiores até os mais elevados no modelo.

Os cinco níveis no modelo CMMI por estágios são mostrados na Figura 26.5. Eles correspondem aos níveis de capacidade de 1 a 5 no modelo contínuo. A diferença fundamental entre os modelos por estágio e contínuo do CMMI é que o modelo por estágios é usado para avaliar a capacidade da organização como um todo, considerando que o modelo contínuo mede a maturidade de áreas de processo específicas dentro da organização.

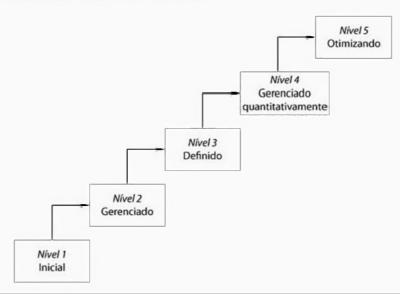
Cada nível de maturidade tem um conjunto associado de áreas de processo e metas genéricas. Estes refletem boas práticas de engenharia e gerenciamento de software e a institucionalização de melhoria de processos. Os níveis de maturidade mais baixos podem ser alcançados por meio da introdução de boas práticas; no entanto, os níveis mais altos requerem um compromisso com a medição e a melhoria de processos.

Por exemplo, as áreas de processo definidas no modelo associado com o segundo nível (o nível gerenciado) são:

1. *Gerenciamento de requisitos*. Gerenciar os requisitos dos produtos e componentes de produto do projeto e identificar as inconsistências entre esses requisitos e os planos de projeto e produtos de trabalho.
2. *Planejamento de projeto*. Estabelecer e manter planos que definem as atividades de projeto.
3. *Monitoração e controle de projeto*. Fornecer a compreensão do progresso do projeto, de maneira que as ações corretivas apropriadas possam ser tomadas quando o desempenho do projeto se desviar significativamente do plano.
4. *Gerenciamento de acordo com fornecedores*. Gerenciar a aquisição de produtos e serviços pelos fornecedores externos ao projeto para os quais existe um acordo formal.
5. *Medição e análise*. Desenvolver e manter a capacidade de medição usada para dar suporte à necessidade de informações da gerência.
6. *Garantia de qualidade de processo e produto*. Fornecer uma visão objetiva para o pessoal e a gerência dos processos e produtos de trabalho associados.
7. *Gerenciamento de configuração*. Estabelecer e manter a integridade dos produtos de trabalho usando identificação de configuração, controle de configuração, *status* de configuração e auditorias de configuração.

Assim como essas práticas específicas, as organizações que operam no segundo nível do modelo CMMI devem alcançar a meta genérica de cada um dos processos de institucionalização como um processo gerenciado. Exemplos de práticas institucionais associadas ao planejamento de projeto que levam o processo de planejamento de projeto a ser um processo gerenciado são:

- Estabelecer e manter uma política organizacional para planejar e executar o processo de planejamento de projeto.
- Fornecer os recursos adequados para a execução do processo de gerenciamento de projeto, desenvolvendo os produtos de trabalho e prestando os serviços do processo.
- Monitorar e controlar o processo de planejamento de projeto contra o plano, além de tomar as ações corretivas apropriadas.

Figura 26.5 O modelo de maturidade CMMI por estágios

- Revisar as atividades, o *status* e os resultados do processo de planejamento de projeto com gerenciamento de alto nível, bem como resolver quaisquer problemas.

A vantagem do CMMI por estágios é que ele é compatível com o modelo de maturidade e de capacidade de software proposto na década de 1980. Muitas empresas compreendem e comprometem-se a usar esse modelo para melhoria de processos. Portanto, é simples para que façam uma transição desse para o modelo CMMI por estágios. Além disso, o modelo por estágios define um caminho de melhoria clara para as organizações. Eles podem planejar se mover do segundo para o terceiro nível e assim por diante.

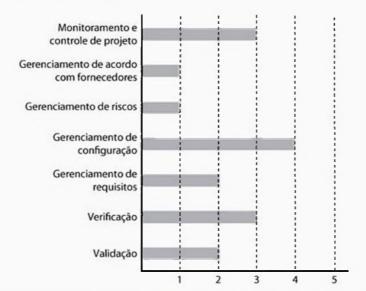
A principal desvantagem do modelo por estágios (e do CMM para Software) é seu caráter prescritivo. Cada nível de maturidade tem suas próprias metas e práticas. O modelo por estágios pressupõe que todas as metas e práticas em um nível sejam implementadas antes da transição para o próximo nível. No entanto, as circunstâncias organizacionais podem ser tais que seja mais adequado implementar as metas e práticas em níveis mais altos antes das práticas de nível mais baixo. Quando uma organização faz isso, uma avaliação da maturidade dá uma imagem enganosa de sua capacidade.

26.5.2 O modelo CMMI contínuo

Os modelos de maturidade contínuos não classificam uma organização de acordo com níveis discretos. Em vez disso, eles são modelos mais refinados que consideram práticas individuais ou grupos de práticas e avaliam o uso de boas práticas dentro de cada grupo de processo. Portanto, a avaliação de maturidade não é um único valor, mas um conjunto de valores mostrando a maturidade da organização para cada processo ou grupo de processos.

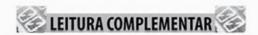
O CMMI contínuo considera as áreas de processo mostradas na Tabela 26.3 e atribui um nível de avaliação de capacidade de 0 a 5 (conforme descrito anteriormente) para cada área de processo. Normalmente, as organizações operam em níveis de maturidade diferentes para áreas de processos diferentes. Consequentemente, o resultado de uma avaliação CMMI contínua é um perfil de capacidade mostrando cada área de processo e sua avaliação de capacidade associada. Um fragmento de um perfil de capacidade que mostra os processos em níveis diferentes de capacidade é mostrado na Figura 26.6. Isso mostra que o nível de maturidade em gerenciamento de configuração, por exemplo, é alto, mas que a maturidade do gerenciamento de riscos é baixa. Uma empresa pode desenvolver perfis de capacidade reais e de alvo em que o perfil-alvo reflete o nível de capacidade que ela gostaria de atingir para essa área de processo.

A principal vantagem do modelo contínuo é que as empresas podem escolher os processos de melhoria de acordo com suas próprias necessidades e requisitos. Em minha experiência, diferentes tipos de organizações têm diferentes requisitos de melhoria de processos. Por exemplo, uma empresa que desenvolve software para a indústria aeroespacial pode concentrar-se em melhorias na especificação de sistema, gerenciamento de configuração

Figura 26.6 Um perfil de capacidade de processos

e validação, considerando que uma empresa de desenvolvimento Web pode estar mais interessada em processos relacionados a clientes. O modelo por estágios obriga as empresas a se concentrarem em diferentes estágios. Por outro lado, o CMMI contínuo permite critério e flexibilidade, ao mesmo tempo que permite às empresas trabalharem no *framework* de melhoria do CMMI.

PONTOS IMPORTANTES

- As metas de melhoria de processos são a qualidade mais elevada de produtos, custos de processos reduzidos e entrega mais rápida de software.
- As principais abordagens para melhoria de processos são abordagens ágeis, orientadas para a redução de *overheads* de processo, e abordagens baseadas em maturidade baseadas no melhor gerenciamento de processos e no uso de boas práticas de engenharia de software.
- O ciclo de melhoria de processo envolve medição de processo, análise e modelagem de processo e mudança de processo.
- Os modelos de processo, que mostram as atividades em um processo e seus relacionamentos com os produtos de software, são usados para a descrição de processo. Na prática, no entanto, os engenheiros envolvidos no desenvolvimento de software sempre adaptam seus modelos para suas circunstâncias locais.
- A medição deve ser usada para responder a questões específicas sobre o processo de software usado. Essas questões devem basear-se em metas de melhoria organizacional.
- Três tipos de métricas de processo usadas no processo de medição são métricas de tempo, métricas de uso de recursos e métricas de eventos.
- O modelo de maturidade de processo CMMI é um modelo de melhoria de processos integrado que suporta a melhoria de processos por estágios e contínua.
- A melhoria de processos no modelo CMMI baseia-se em atingir um conjunto de metas relacionadas às boas práticas de engenharia de software e em descrever, padronizar e controlar as práticas usadas para atingir essas metas. O modelo CMMI inclui práticas recomendadas que podem ser usadas, mas não são obrigatórias.

LEITURA COMPLEMENTAR

'Can you trust software capability evaluations?' Esse artigo tem um olhar cético sobre o objeto de avaliação de capacidade, em que a maturidade de processo da empresa é avaliada, e discute por que essas avaliações podem não dar uma imagem verdadeira de maturidade de uma organização. (O'CONNELL, E.; SAIEDIAN, H. *IEEE Computer*, v. 33, n. 2, fev. 2000.) Disponível em: <http://dx.doi.org/10.1109/2.820036>.

Software Process Improvement: Results and Experience from the Field. Esse livro é uma coleção de artigos centrados em estudos de caso de melhoria de processos em várias pequenas e médias empresas norueguesas. Ele também inclui uma boa introdução às questões gerais de melhoria de processos. (CONRADI, R.; DYBÅ, T.; SJØBERG, D.; ULSUND, T. (Orgs.). *Software Process Improvement*: Results and Experience from the Field. Springer, 2006.)

CMMI: Guidelines for Process Integration and Product Improvement, 2nd edition. Uma descrição abrangente do CMMI. O CMMI é grande e complexo, e é praticamente impossível tornar a leitura e compreensão fáceis. Esse livro faz um trabalho razoável, incluindo algum material anedótico e histórico, mas, ainda assim, às vezes, é difícil. (CHRISSIS, M. B.; KONRAD, M.; SHRUM S. *CMMI: Guidelines for Process Integration and Product Improvement.* Addison-Wesley, 2007.)

EXERCÍCIOS

26.1 Quais são as diferenças importantes entre a abordagem ágil e a abordagem de maturidade de processo para a melhoria de processos de software?

26.2 Em quais circunstâncias a qualidade de produto pode ser determinada pela qualidade da equipe de desenvolvimento? Dê exemplos de tipos de produtos de software que são particularmente dependentes de talentos e capacidades individuais.

26.3 Sugira três ferramentas de software especializadas que podem ser desenvolvidas para apoiar um programa de melhoria de processos em uma organização.

26.4 Assuma que a meta da melhoria de processos em uma organização seja aumentar o número de componentes reusáveis produzidos durante o desenvolvimento. Sugira três questões no paradigma GQM que possam conduzir para essa meta.

26.5 Descreva três tipos de métricas de processo de software que podem ser coletados como parte de um processo de melhoria de processos. Dê um exemplo de cada tipo de métrica.

26.6 Projete um processo para a avaliação e priorização de propostas de mudança de processos. Documente-o como um modelo de processo mostrando os papéis envolvidos nele. Você deve usar diagramas de atividade de UML ou BPMN para descrever o processo.

26.7 Dê duas vantagens e duas desvantagens da abordagem para a avaliação e melhoria de processo embutidas nos *frameworks* de melhoria de processos, como o CMMI.

26.8 Sob quais circunstâncias você recomendaria o uso da representação por estágios do CMMI?

26.9 Quais são as vantagens e as desvantagens do uso de um modelo de maturidade de processo que se concentra nas metas a serem atingidas, em vez de nas boas práticas a serem introduzidas?

26.10 Você acha que programas de melhoria de processos, que envolvem mensurar o trabalho de pessoas no processo e a introdução de mudanças nesse processo, podem ser inerentemente desumanizados? Qual resistência pode surgir a um programa de melhoria de processos e por quê?

REFERÊNCIAS

AHERN, D. M.; CLOUSE, A.; TURNER, R. *CMMI Distilled*. Reading, Mass.: Addison-Wesley, 2001.

BASILI, V.; GREEN, S. Software Process Improvement at the SEL. *IEEE Software*, v. 11, n. 4, 1993, p. 58-66.

BASILI, V. R.; ROMBACH, H. D. The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Trans. on Software Eng.*, v. 14, n. 6, 1988, p. 758-773.

BATE, R. A Systems Engineering Capability Maturity Model Version 1.1. *Software Engineering Institute*, 1995.

CHRISSIS, M. B.; KONRAD, M.; SHRUM, S. *CMMI*: Guidelines for Process Integration and Product Improvement. 2. ed. Boston: Addison-Wesley, 2007.

CURTIS, B.; HEFLEY, W. E.; MILLER, S. A. *The People Capability Model*: Guidelines for Improving the Workforce. Boston: Addison-Wesley, 2001.

HAASE, V.; MESSNARZ, R.; KOCH, G.; KUGLER, H. J.; DECRINIS, P. Bootstrap: Fine Tuning Process Assessment. *IEEE Software*, v. 11, n. 4, 1994, p. 25-35.

HAMMER, M. Reengineering Work: Don't Automate, Obliterate. *Harvard Business Review*, jul.-ago. 1990, p. 104-112.

HUMPHREY, W. *Managing the Software Process*. Reading, Mass.: Addison-Wesley, 1989.

_______. Characterizing the Software Process. *IEEE Software*, v. 5, n. 2, 1988, p. 73-79.

_______. *A Discipline for Software Engineering*. Reading, Mass.: Addison-Wesley, 1995.

KUVAJA, P.; SIMILÄ, J.; KRZANIK, L.; BICEGO, A.; SAUKKONEN, S.; KOCH, G. *Software Process Assessment and Improvement*: The BOOTSTRAP Approach. Oxford: Blackwell Publishers, 1994.

OSTERWEIL, L. Software Processes are Software Too. 9th Int. Conf. on Software Engineering, *IEEE Press*, 1987, p. 2-12.

OULD, M. A. *Business Processes*: Modeling and Analysis for Re-engineering and Improvement. Chichester: John Wiley & Sons, 1995.

PAULK, M. C.; CURTIS, B.; CHRISSIS, M. B.; WEBER, C. V. Capability Maturity Model, Version 1.1. *IEEE Software*, v. 10, n. 4, 1993, p. 18-27.

PAULK, M. C.; KONRAD, M. An Overview of ISO's SPICE Project. *IEEE Computer*, v. 27, n. 4, 1994, p. 68-70.

PAULK, M. C.; WEBER, C. V.; CURTIS, B.; CHRISSIS, M. B. *The Capability Maturity Model*: Guidelines for Improving the Software Process. Reading, Mass.: Addison-Wesley, 1995.

PULFORD, K.; KUNTZMANN-COMBELLES, A.; SHIRLAW, S. *A Quantitative Approach to Software Management*. Wokingham: Addison-Wesley, 1996.

SOMMERVILLE, I.; SAWYER, P. *Requirements Engineering*: A Good Practice Guide. Chichester: John Wiley & Sons, 1997.

WHITE, S. A. An Introduction to BPMN. 2004. Disponível em: <http://www.bpmn.org/Documents/Introduction%20to%20BPMN>.

Glossário

Ada
Linguagem de programação que foi desenvolvida para o Departamento de Defesa dos EUA na década de 1980 como linguagem padrão para desenvolvimento de software militar. Está baseada nas pesquisas sobre linguagens de programação da década de 1970 e inclui construções como tipos abstratos de dados e apoio para concorrência. Ainda é usada para grandes e complexos sistemas militares e aeroespaciais.

análise estática
Análise baseada em ferramentas de um código-fonte de programa para descobrir defeitos e anomalias. As anomalias, como atribuições sucessivas de uma variável sem nenhum uso intermediário, podem ser erros de programação.

arquitetura cliente-servidor
Um modelo de arquitetura para sistemas distribuídos onde a funcionalidade do sistema é disponibilizada como um conjunto de serviços fornecidos por um servidor. Esses serviços são acessados e usados pelos computadores clientes. Variações dessa abordagem, tal como arquitetura cliente-servidor de três camadas, usam múltiplos servidores.

arquitetura de referência
Uma arquitetura genérica e idealizada que inclui todos os recursos que os sistemas podem incorporar. É uma maneira de informar os projetistas sobre a estrutura geral dessa classe de sistema, mais do que apenas uma base para a criação de uma arquitetura de sistema específica.

arquitetura de software
Modelo de estrutura e organização fundamentais de um sistema de software.

arquitetura dirigida a modelos (MDA – *model-driven architecture*)
Uma abordagem para desenvolvimento de software baseada na construção de um conjunto de modelos de sistema e que pode ser processada automática ou semiautomaticamente para gerar um sistema executável.

ataque de negação de serviço
Um ataque em um software baseado na Internet que tenta sobrecarregar o sistema de tal forma que este não consiga fornecer seu serviço normal para usuários.

BEA
Um fornecedor de sistemas ERP norte-americano.

bomba de insulina
Dispositivo médico controlado por software que pode fornecer doses controladas de insulina para as pessoas que sofrem de diabetes. Usado como estudo de caso em vários capítulos deste livro.

BPMN
Business Process Modeling Notation (Notação para modelagem de processos de negócios). Uma notação para definição de *workflows*.

C
Uma linguagem de programação originalmente desenvolvida para implementar sistemas Unix. C é uma linguagem relativamente de baixo nível de implementação de sistema e que permite acesso ao hardware do sistema e que pode ser compilada para um código eficiente. É muito usada para programação de baixo nível de sistemas e para desenvolvimento de sistemas embutidos.

C++
Uma linguagem de programação orientada a objetos que é um superconjunto de C.

C#
Uma linguagem de programação orientada a objetos desenvolvida pela Microsoft e que tem muito em comum com C++, mas que inclui recursos que permitem mais verificações de tipo em tempo de compilação.

CASE (*Computer-Aided Software Engineering*)
Engenharia de software auxiliada por computador. O processo de desenvolvimento de software com uso de suporte automatizado.

CASE *workbench*
Um conjunto integrado de ferramentas CASE que trabalham em conjunto para apoiar uma grande atividade do processo, como projeto de software ou gerenciamento de configuração.

caso de confiança
Um documento estruturado usado para guardar as definições feitas por um desenvolvedor de sistemas a respeito da confiança de um sistema.

caso de segurança
Declaração estruturada de que um sistema é seguro. Muitos sistemas críticos devem ter casos de segurança associados que sejam avaliados e aprovados pelos reguladores externos, antes que o sistema seja certificado para uso.

caso de uso
Especificação de um tipo de interação com o sistema.

cenário
Descrição de uma maneira típica de como o sistema é usado ou de como um usuário realizou alguma atividade.

ciclo de vida de software
Frequentemente usado com outro nome para o processo de software; originalmente inventado para se referir ao modelo em cascata de processo de software.

classe de objeto
Uma classe de objeto define os atributos e as operações dos objetos. Objetos são criados em tempo de execução ao instanciar a definição da classe. O nome da classe de objeto pode ser usado como um nome de tipo em algumas linguagens orientadas a objeto.

CMM
Modelo de maturidade e de capacidade do Software Engineering Institute que é usado para avaliar o nível de maturidade do desenvolvimento de software dentro de uma organização. Foi substituído pelo CMMI, mas ainda é bastante usado.

CMMI
Abordagem integrada para a modelagem da maturidade e de capacidade de processo baseada na adoção de boas práticas de engenharia de software e no gerenciamento integrado de qualidade. Ela apoia a modelagem de maturidade discreta e contínua, além de integrar modelos de maturidade de processos de engenharia de sistemas e de software.

cobertura de testes
Eficácia dos testes do sistema ao testar o código do sistema inteiro. Algumas empresas possuem padrões para cobertura de testes (por exemplo, os testes de sistema devem garantir que todas as instruções do programa sejam executadas pelo menos uma vez).

código de ética e prática profissional
Um conjunto de diretrizes que define o comportamento ético e profissional esperado pelos engenheiros de software. Isso foi definido pelas principais sociedades profissionais dos EUA (ACM e IEEE) e estabelece o comportamento ético em oito tópicos: público, cliente e empregador, produto, julgamento, gerenciamento, colegas, profissão e o profissional em si.

COM+
Modelo de componentes e *middleware* projetado para ser usado em plataformas da Microsoft; atualmente substituído pelo .NET.

componente
Uma unidade de software independente e implantável e que é completamente definida e acessada através de um conjunto de interfaces.

compositor de aspectos
Um programa que é normalmente parte de um sistema de compilação que processa um programa orientado a aspectos e modifica o código para incluir aspectos definidos nos pontos específicos do programa.

computação em nuvem
Fornecimento de computação e/ou serviços de aplicação através da Internet usando uma 'nuvem' de servidores a partir de um provedor externo. A 'nuvem' é implementada com o uso de um grande número de computadores e tecnologias de virtualização para fazer uso efetivo desses sistemas.

confiabilidade
A capacidade do sistema de fornecer os serviços conforme especificado. Confiabilidade pode ser expressa quantitativamente como sendo a probabilidade de uma falha sob demanda ou como a taxa de ocorrência de falha.

confiança
A confiança de um sistema é uma propriedade agregada que leva em conta segurança, confiabilidade, disponibilidade, proteção e outros atributos do sistema. A confiança de um sistema reflete o quanto os usuários podem confiar nele.

construção de sistemas
O processo de compilação de componentes ou unidades que compõem o sistema e a ligação desses com outros componentes para criar um programa executável. Construção de sistema é normalmente automatizada para que recompilação seja minimizada. Essa automação pode estar embutida no sistema de processamento da linguagem (Java) ou envolver ferramentas de software para auxiliar na construção de sistema.

CORBA (*Common Request Broker Architecture*)
Um conjunto de padrões proposto por Object Management Group (OMG) que define modelos de objetos distribuídos e comunicações entre objetos; influente no desenvolvimento de sistemas distribuídos, mas raramente usado na atualidade.

CVS
Ferramenta de código fonte aberto amplamente usada para gerenciamento de versões.

desenvolvimento de software orientado a aspectos
Uma abordagem de desenvolvimento de software que combina o desenvolvimento baseado em geradores e em componentes. Interesses transversais são identificados em um programa e a implementação desses interesses é definida como aspectos. Aspectos incluem uma definição sobre onde devem ser incorporados dentro de um programa. Um compositor de aspectos, então, compõe os aspectos em locais apropriados dentro do programa.

desenvolvimento dirigido a modelos (MDD – *model-driven development*)
Uma abordagem para engenharia de software focada em modelos de sistema expressos através de UML, em vez de um código de linguagem de programação. Isso estende MDA para considerar outras atividades além do desenvolvimento, como engenharia de requisitos e testes.

desenvolvimento dirigido a testes
Uma abordagem para desenvolvimento de software onde os testes executáveis são escritos antes do código do programa. O conjunto de testes é executado automaticamente depois de cada alteração feita no programa.

desenvolvimento incremental
Uma abordagem para desenvolvimento de software onde este é entregue e implantado em incrementos.

desenvolvimento iterativo
Uma abordagem para desenvolvimento de software onde os processos de especificação, projeto, programação e teste são intercalados.

desenvolvimento orientado a objetos (OO)
Uma abordagem para desenvolvimento de software onde as principais abstrações no sistema são objetos independentes. O mesmo tipo de abstração é usado durante especificação, projeto e desenvolvimento.

desenvolvimento rápido de aplicações (RAD – *rapid application development*)
Uma abordagem para desenvolvimento de software com objetivo de entregar software rapidamente. Comumente envolve o uso de ferramentas de programação de banco de dados e de apoio ao desenvolvimento, como geradores de telas e relatórios.

detecção de defeitos
Uso de processos e verificações em tempo de execução para detectar e remover defeitos em um programa, antes que resultem em falhas do sistema.

diagrama de atividades (PERT)
Diagrama usado pelos gerentes de projeto para mostrar as dependências entre as tarefas que devem ser concluídas. O diagrama mostra as tarefas, o tempo esperado para a conclusão delas e suas dependências. O caminho crítico é o caminho mais longo (em termos de tempo necessário para concluir as tarefas) através do diagrama de atividades. O caminho crítico define o tempo mínimo requerido para concluir o projeto.

diagrama de barras
Um diagrama usado por gerentes de projeto para mostrar as tarefas do projeto, o cronograma associado com essas tarefas e as pessoas que trabalharão nelas. Ele mostra as datas inicial e final das tarefas e a alocação do pessoal na linha de tempo.

diagrama de classes
Um tipo de diagrama UML mostra as classes de objetos em um sistema e seus relacionamentos.

diagrama de estado
Diagrama UML que mostra os estados de um sistema e os eventos que ativam a transição de um estado para outro.

diagrama de sequência
Diagrama que mostra a sequência de interações requeridas para completar alguma operação. Em UML, diagramas de sequência podem estar associados com casos de uso.

dinâmica de evolução de programa
O estudo das formas de como as mudanças de um sistema de software evoluem. Afirma-se que as leis de Lehman governam a dinâmica de evolução de programa.

disponibilidade
A prontidão de um sistema para fornecer serviços quando requisitado. A disponibilidade é normalmente expressa como um número decimal, assim uma disponibilidade de 0,999 significa que o sistema pode fornecer serviços para 999 de 1.000 unidades de tempo.

domínio
Um problema específico ou área de negócio em que os sistemas de software são usados. Exemplos de domínios incluem controle em tempo real, processamento de dados corporativos e comutação em telecomunicações.

DSDM
Dynamic System Development Method; tido como um dos primeiros métodos ágeis de desenvolvimento.

engenharia de software baseada em componentes (CBSE – *component-based software engineering*)
Desenvolvimento de software através da composição de componentes de software independentes e implantáveis que estão consistentes com um modelo de componentes.

engenharia de sistemas
Processo cujo foco é a especificação de um sistema, a integração de seus componentes e os testes para verificar se o sistema atende a seus requisitos. A engenharia de sistemas preocupa-se com o sistema sociotécnico inteiro – software, hardware e processos operacionais – e não apenas com o sistema de software.

engenharia de software *cleanroom*
Uma abordagem para desenvolvimento de software onde o objetivo é evitar a introdução de defeitos (em analogia com uma sala limpa usada na fabricação de semicondutores). O processo envolve especificação formal de software, transformação estruturada de uma especificação para um programa, desenvolvimento de argumentos de correção e testes estatísticos de programa.

***enterprise Java beans* (EJB)**
Modelo de componentes baseado em Java.

etnografia
Uma técnica de observação que pode ser usada na elicitação e análise de requisitos. O etnógrafo imerge no ambiente dos usuários e observa os hábitos de seu trabalho diário. Os requisitos de software podem ser deduzidos a partir dessas observações.

Extreme Programming (XP)
Método ágil de desenvolvimento de software amplamente usado que inclui práticas como requisitos baseados em cenários, desenvolvimento *test-first* e programação em pares.

família de aplicações
Um conjunto de programas de aplicação que possuem uma arquitetura comum e funcionalidade genérica. Estes podem ser adaptados às necessidades de clientes específicos, alterando componentes e parâmetros do programa.

ferramenta CASE
Uma ferramenta de software, como um editor de projeto ou um *debugger* (depurador) de programas, usada para apoiar uma atividade dentro do processo de desenvolvimento de software.

***framework* de aplicações**
Um conjunto de classes abstratas e concretas reusáveis que implementam recursos comuns para muitas aplicações dentro de um domínio (por exemplo, interface de usuário). As classes do *framework* de aplicações são especializadas e instanciadas para criar uma aplicação.

garantia de qualidade (QA – *quality assurance*)
O processo geral para definir como a qualidade de software pode ser alcançada e como a organização que desenvolve o software sabe que ele atingiu o nível de qualidade requerido.

gerador de programas
Um programa que gera outros programas a partir de uma especificação abstrata de alto nível. O gerador contém o conhecimento que é reusado em cada atividade de geração.

gerenciamento de configuração
O processo de gerenciamento de mudanças de um produto de software em evolução. Gerenciamento de configuração envolve planejamento de configuração, gerenciamento de versões, construção de sistemas e gerenciamento de mudanças.

gerenciamento de mudanças
Processo para registrar, verificar, analisar, estimar e implementar mudanças propostas para um sistema de software.

gerenciamento de requisitos
O processo de gerenciamento de mudanças nos requisitos para garantir que as mudanças feitas sejam analisadas adequadamente e acompanhadas através do sistema.

gerenciamento de riscos
O processo de identificação de riscos, avaliação de sua severidade e planejamento de medidas a serem tomadas caso o risco ocorra; monitoramento de software e processo de software para riscos.

gerenciamento de versões
O processo de gerenciamento de mudanças feitas em um sistema de software e seus componentes, para que seja possível saber quais mudanças foram implementadas em cada versão do componente/sistema e também para recuperar/recriar versões anteriores do componente/sistema.

gráfico de Gantt
Um nome alternativo para diagrama de barras.

inspeção
Veja *inspeção de programa*.

inspeção de programa
Uma revisão onde um grupo de inspetores examina um programa, linha por linha, com o objetivo de detectar defeitos. As inspeções são comumente guiadas por uma lista de verificação de defeitos de programação comuns.

interface
Uma especificação de atributos e operações associados com um componente de software. A interface é usada como meio para acessar a funcionalidade do componente.

Interface de Programa de Aplicação (API – *Application Program Interface*)
Uma interface, normalmente especificada como um conjunto de operações, que permite acesso a uma funcionalidade da aplicação. Isso significa que essa funcionalidade pode ser chamada diretamente por outros programas e não apenas acessada através da interface de usuário.

ISO 9000/9001
Conjunto de normas para processos de gerenciamento da qualidade definido pela International Organization for Standardization (ISO). ISO 9001 é a norma da ISO mais aplicável para desenvolvimento de software. Pode ser usado para certificar processos de gerenciamento da qualidade em uma organização.

item de configuração
Uma unidade que pode ser lida por máquina, como um documento ou um arquivo de código-fonte, que está sujeita às mudanças e onde estas devem ser controladas por um sistema de gerenciamento de configuração.

J2EE
Java 2 Platform Enterprise Edition. Um sistema complexo de *middleware* que suporta o desenvolvimento em Java de aplicações web baseadas em componentes. Inclui modelos de componentes para componentes Java, APIs, serviços etc.

Java
Linguagem de programação orientada a objetos amplamente usada e que foi projetada pela Sun com o objetivo de ser independente de plataforma.

leis de Lehman
Um conjunto de hipóteses sobre os fatores que influenciam a evolução de sistemas complexos de software.

linguagem de restrição de objetos (OCL – *object constraint language*)
Uma linguagem que faz parte de UML, usada para definir predicados que se aplicam a classes de objetos e interações em um modelo UML. O uso de OCL para especificar componentes é parte fundamental do desenvolvimento dirigido a modelos.

linha de produtos de software
Veja *família de aplicações*.

make
Uma das primeiras ferramentas de construção de sistemas; ainda amplamente usada em sistemas Unix/Linux.

manifesto ágil
Um conjunto de princípios que encapsula as ideias que apoiam métodos ágeis de desenvolvimento de software.

manutenção
O processo de realizar mudanças em um sistema depois que o mesmo foi colocado em produção.

melhoria de processo
A realização de mudanças em um processo de desenvolvimento de software com o propósito de torná-lo mais previsível ou aumentar a qualidade de suas saídas. Por exemplo, se seu propósito é reduzir o número de defeitos no software entregue, você poderia melhorar o processo pela adição de novas atividades de validação.

método estruturado
Método de projeto de software que define os modelos de sistema que devem ser desenvolvidos, as regras e as diretrizes que devem ser aplicadas a esses modelos e um processo a ser seguido durante o desenvolvimento do projeto.

métodos ágeis
Métodos de desenvolvimento de software que são orientados à entrega rápida de software. O software é desenvolvido e entregue incrementalmente e a documentação e burocracia do processo são minimizadas. O foco do desenvolvimento está no código em si, em vez de na documentação de apoio.

métodos formais
Métodos de desenvolvimento de software, onde ele é modelado usando construções matemáticas formais como predicados e conjuntos. Transformação formal converte esse modelo para código. Normalmente usado na especificação e desenvolvimento de sistemas críticos.

métrica de previsão
Uma métrica de software usada como base para fazer previsões sobre as características de um sistema de software, como sua confiabilidade e manutenibilidade.

métrica de software
Um atributo de um sistema ou processo de software que pode ser expresso numericamente e medido. As métricas de processo são atributos do processo, tal como tempo necessário para completar uma tarefa; as métricas do produto são atributos do software em si, tal como tamanho ou complexidade.

métricas de controle
Métrica de software que permite aos gerentes tomar decisões de planejamento com base nas informações sobre o processo de software ou o produto de software que está sendo desenvolvido. A maioria das métricas de controle é métricas de processo.

MHC-PMS
Mental Health Care Patient Management System; usado como estudo de caso em vários capítulos.

middleware
Software de infraestrutura em um sistema distribuído. Ele ajuda a gerenciar interações entre entidades distribuídas do sistema e bancos de dados do sistema. Exemplos de *middleware* são um localizador de objetos e um sistema de gerenciamento de transações.

modelagem algorítmica de custos
Uma abordagem para estimar custo de software, onde uma fórmula é usada para estimar o custo do projeto. Os parâmetros da fórmula são atributos do projeto e do software em si.

Modelagem Construtiva de Custos (COCOMO – *Constructive Cost Modeling*)
Um conjunto de modelos algorítmicos para estimativas de custos. COCOMO foi proposto pela primeira vez no começo da década de 1980 e foi modificado e atualizado desde então, para refletir novas tecnologias e práticas de engenharia de software em constante mudança.

modelagem de crescimento de confiabilidade
Desenvolvimento de um modelo sobre como muda (melhora) a confiabilidade de um sistema à medida que o mesmo é testado e os defeitos do programa são removidos.

modelo de componentes
Um conjunto de padrões para implementação, documentação e implantação de componentes. Esses padrões abrangem interfaces específicas que podem ser fornecidas por um componente, denominação, interoperação e composição de componentes. Modelos de componentes fornecem a base para o *middleware* apoiar a execução de componentes.

modelo de componentes CORBA
Um modelo de componentes projetado para ser usado em plataformas CORBA.

modelo de domínio
Definição de abstrações do domínio, como políticas, procedimentos, objetos, relacionamentos e eventos. Serve como base de conhecimento sobre alguma área de negócio.

modelo de maturidade de processo
Modelo extenso no qual um processo inclui boas práticas e capacidades de ponderação e medição conduzidas para a melhoria de processo.

Modelo de Maturidade e de Capacidade de Pessoas (P-CMM – *People Capability Maturity Model*)
Modelo de maturidade de processo que reflete o grau de eficiência de uma organização no gerenciamento de habilidades, treinamento e experiência das pessoas nessa organização.

modelo de objeto
Modelo de um sistema de software que é estruturado e organizado como um conjunto de classes de objeto e relações entre essas classes. Várias perspectivas diferentes do modelo podem existir, como perspectiva de estado e de sequência.

modelo de processo
Representação abstrata de um processo. Os modelos de processo podem ser desenvolvidos a partir de várias perspectivas e podem mostrar as atividades envolvidas em um processo, os artefatos usados, as restrições que se aplicam a ele e os papéis das pessoas que o aprovam.

modelo em cascata
Modelo de processo de software que envolve estágios discretos de desenvolvimento: especificação, projeto, implementação, teste e manutenção. A princípio, um estágio deve ser completado antes que seja possível avançar para o próximo. Na prática, há uma iteração significativa entre os estágios.

modelo espiral
Modelo de um processo de desenvolvimento onde o processo é representado como uma espiral, onde cada volta incorpora diferentes estágios no processo. À medida que se passa de uma volta para outra, são repetidos todos os estágios do processo.

.NET
Um *framework* muito extenso utilizado para desenvolver aplicações para os sistemas Microsoft Windows; inclui um modelo de componente que define padrões para componentes nos sistemas Windows e *middleware* associado para suportar a execução de componentes.

Object Management Group (OMG)
Um grupo de empresas formado para definir padrões para desenvolvimento orientado a objetos. Exemplos de padrões definidos pela OMG são CORBA, UML e MDA.

ocultação das informações
Uso de construções de linguagens de programação para ocultar a representação das estruturas de dados e controlar o acesso externo a essas estruturas.

open source
Uma abordagem para desenvolvimento de software onde o código-fonte de um sistema é tornado público e usuários externos são encorajados a participar do desenvolvimento do sistema.

padrão de arquitetura (estilo)
Uma descrição abstrata de uma arquitetura de software que foi provada e testada em certo número de sistemas de software diferentes. A descrição do padrão inclui informações sobre onde é adequado o uso do padrão e da organização dos componentes da arquitetura.

padrão de projeto
Uma solução bem provada para um problema comum que captura a experiência e as boas práticas de forma que possa ser reusada. É uma representação abstrata que pode ser instanciada de várias maneiras.

plano de qualidade
Um plano que define os processos e os procedimentos de qualidade que devem ser usados. Isso envolve selecionar e instanciar padrões para produtos e processos e definir atributos de qualidade do sistema que são mais importantes.

prevenção a defeitos
Desenvolver software de tal forma que os defeitos não sejam introduzidos nele.

probabilidade de falha sob demanda (POFOD – *probability of failure on demand*)
Uma métrica de confiabilidade que é baseada na possibilidade de um sistema de software falhar quando é feita uma requisição por seus serviços.

processo de software
Conjunto de atividades e processos relacionados envolvidos no desenvolvimento e na evolução de um sistema de software.

programação em pares
Uma prática de desenvolvimento onde os programadores trabalham em pares, em vez de individualmente, para desenvolver o código; uma parte fundamental de Extreme Programming.

projeto de interface de usuário
O processo de projetar a maneira como os usuários podem acessar a funcionalidade do sistema e a maneira como as informações produzidas por ele são exibidas.

propriedade emergente
Uma propriedade que aparece somente quando todos os componentes foram integrados para criar o sistema.

proteção
A capacidade de um sistema de proteger a si mesmo contra intrusões acidentais ou intencionais. Proteção envolve confidencialidade, integridade e disponibilidade.

Python
Uma linguagem de programação com tipos dinâmicos e que se adapta muito bem para o desenvolvimento de sistemas web; muito usada pela Google.

***Rational Unified Process* (RUP)**
Um modelo genérico de processo de software que apresenta o desenvolvimento de software como uma atividade iterativa de quatro fases: concepção, elaboração, construção e transição. A concepção estabelece um caso de negócio para o sistema, a elaboração define a arquitetura, a construção implementa o sistema e a transição implanta o sistema no ambiente do cliente.

reengenharia
Modificação de um sistema de software para torná-lo mais fácil de ser entendido e alterado. Reengenharia muitas vezes envolve reestruturação e organização de software e de dados, simplificação de programas e redocumentação.

reengenharia, processo de negócio
Mudança do processo de negócio para atender a um novo objetivo da organização, como redução de custos e execução mais rápida.

release
Versão de um sistema de software que é disponibilizada para os clientes do sistema.

requisito de confiança
Um requisito de sistema incluso para ajudar a alcançar a confiança requerida para um sistema. Os requisitos não funcionais de confiança especificam os valores dos atributos de confiança; requisitos funcionais de confiança são requisitos que especificam como evitar, detectar, tolerar ou recuperar defeitos e falhas de sistema.

requisito funcional
Declaração de uma função ou de uma característica que deve ser implementada no sistema.

requisito não funcional
Declaração de uma restrição ou de um comportamento esperado que se aplica ao sistema. Essa restrição pode se referir às propriedades emergentes do software que está sendo desenvolvido ou ao processo de desenvolvimento.

REST
REST é sigla para *Representational State Transfer*, que é um estilo de desenvolvimento baseado em interação simples de cliente/servidor e que usa o protocolo HTTP. REST é baseado na ideia de recurso identificável, o qual possui uma URI. Todas as interações com recursos são baseadas em HTTP POST, GET, PUT e DELETE. Atualmente é muito usado para implementar *web services* de baixo *overhead*.

risco
Uma condição indesejável que representa ameaça para que algum objetivo seja alcançado. Um risco de processo ameaça o cronograma ou o custo de um processo; um risco de produto pode significar que alguns dos requisitos de software podem não ser alcançados.

Ruby
Uma linguagem de programação com tipos dinâmicos e que se adapta muito bem para programação de aplicações web.

SAP
Uma empresa alemã desenvolvedora de um sistema ERP muito conhecido. A palavra se refere também ao nome do próprio sistema ERP.

Scrum
Um método ágil de desenvolvimento baseado em *sprints* – ciclos curtos de desenvolvimento. Scrum pode ser usado como base para gerenciamento ágil de projetos juntamente com outros métodos ágeis, como XP.

segurança
A capacidade de um sistema de operar sem falha catastrófica.

SEI
Software Engineering Institute. Um centro de pesquisa sobre engenharia de software e transferência de tecnologia fundado com o propósito de aprimorar o padrão de engenharia de software em empresas dos EUA.
serviço
Veja *web service*.
servidor
Programa que fornece um serviço para outros programas (clientes).
sistema crítico
Um sistema computacional cuja falha pode resultar em perdas econômicas, humanas ou ambientais significativas.
sistema de processamento de dados
Sistema cujo objetivo é processar grandes quantidades de dados estruturados. Esse tipo de sistema normalmente processa dados em lotes e segue um modelo entrada-processo-saída. Exemplos de sistemas de processamento de dados são sistemas de faturamento e sistemas de pagamentos.
sistema de processamento de linguagens
Sistema que traduz uma linguagem para outra. Por exemplo, um compilador é um sistema de processamento de linguagens que traduz o código-fonte de um programa para código-objeto.
sistema de processamento de transações
Sistema que garante que as transações sejam processadas de tal forma que não interfiram umas com as outras e que a falha de uma transação individual não afete outras transações ou dados do sistema.
sistema de tempo real
Um sistema que deve reconhecer e processar eventos externos em "tempo real". A correção do sistema não depende apenas do que ele faz, mas também de quão rápido ele faz. Sistemas de tempo real são, normalmente, organizados como um conjunto de processos sequenciais cooperativos.
sistema distribuído
Um sistema de software onde os subsistemas ou componentes de software executam em processadores diferentes.
sistema embutido
Um sistema de software que está embutido em um dispositivo de hardware (por exemplo, o sistema de software em um telefone celular). Os sistemas embutidos são normalmente sistemas de tempo real e precisam responder em tempos compatíveis aos eventos ocorridos em seu ambiente.
sistema ERP (*enterprise resource planning*)
Um sistema de software de grande porte que inclui uma variedade de recursos para apoiar a operação dos negócios das empresas e que fornece meios para compartilhar informação entre esses recursos. Por exemplo, um sistema ERP pode incluir apoio para gerenciamento da cadeia de suprimentos, manufatura e distribuição. Os sistemas ERP são configurados para requisitos de cada empresa que utiliza o sistema.
sistema legado
Um sistema sociotécnico que é útil ou até essencial para uma organização, mas que foi desenvolvido com uso de tecnologias ou métodos obsoletos. Pelo fato de os sistemas legados frequentemente executarem funções críticas para o negócio, eles precisam ser mantidos.
sistema meteorológico no deserto
Sistema para coletar dados sobre as condições de tempo em áreas remotas. Usado como estudo de caso em vários capítulos deste livro.
sistema ponto a ponto
Um sistema distribuído onde não há distinção entre servidores e clientes. Computadores do sistema podem agir como clientes ou servidores. Aplicações ponto a ponto incluem compartilhamento de arquivos, programas de mensagens instantâneas e sistemas de apoio à cooperação.
sistema sociotécnico
Um sistema, incluindo componentes de hardware e software, que tem definidos os processos operacionais seguidos de operadores humanos e que funciona dentro de uma organização. Ele é, portanto, influenciado pelas políticas, procedimentos e estruturas organizacionais.
sistemas baseados em eventos
Sistemas nos quais o controle de operação é determinado por eventos que são gerados no ambiente do sistema. A maioria dos sistemas de tempo real é de sistemas baseados em eventos.
***Structured Query Language* (SQL)**
Linguagem padrão usada para programação de bancos de dados relacionais.

Subversion
Ferramenta *open source* de construção de sistemas que está disponível em diversas plataformas.
taxa de ocorrência de falhas (ROCOF – *rate of occurence of failure*)
Uma métrica de confiabilidade baseada no número de falhas observadas de um sistema dentro de um dado período de tempo.
tempo médio para falha (MTTF – *mean time to failure*)
O tempo médio observado entre as falhas do sistema; usado na especificação de confiabilidade.
testes de caixa branca
Uma abordagem para testes de programas onde os testes são baseados no conhecimento da estrutura do programa e de seus componentes. Acesso ao código-fonte é essencial para testes de caixa branca.
testes de caixa preta
Uma abordagem de testes onde os testadores não têm acesso ao código-fonte do sistema ou seus componentes. Os testes são derivados da especificação do sistema.
tipo abstrato de dado
Um tipo que é definido por suas operações em vez de sua representação. A representação é privada e somente pode ser acessada pelas operações definidas.
tolerância a defeitos
Capacidade de um sistema de continuar a execução mesmo depois da ocorrência de defeitos.
transação
Uma unidade de interação com um sistema computacional. As transações são independentes e atômicas (não são divididas em unidades menores) e são uma unidade fundamental de recuperação, consistência e concorrência.
***Unified Modeling Language* (UML)**
Linguagem gráfica usada no desenvolvimento orientado a objetos que inclui diversos tipos de modelos de sistema que fornecem visões diferentes de um sistema. UML se tornou um padrão de fato para modelagem orientada a objetos.
validação
O processo de verificar se o sistema atende às necessidades e às expectativas do cliente.
verificação
O processo de verificar se o sistema atende a sua especificação.
verificação de modelo
Um método de verificação estática onde um modelo de estado de um sistema é analisado exaustivamente, na tentativa de descobrir estados inalcançáveis.
web service
Um componente de software independente que pode ser acessado através da Internet usando protocolos padronizados. Ele é completamente autocontido e sem dependências externas. Foram desenvolvidos padrões baseados em XML, como SOAP (*Standard Object Access Protocol*), para troca de informações com *web services* e WSDL (*Web Service Definition Language*), para definição de interfaces de *web services*. Contudo, a abordagem REST também pode ser usada para implementação de *web services*.
workflow
Definição detalhada de um processo de negócio que se destina à execução de uma determinada tarefa. *Workflow* é normalmente expresso graficamente e mostra as atividades de processos individuais e a informação que é produzida e consumida por cada atividade.
WSDL
Notação baseada em XML para definir a interface de *web services*.
XML
Extended Markup Language. XML é uma linguagem de marcação de texto que suporta a troca de dados estruturados. Cada campo de dados é delimitado por *tags* que fornecem informação sobre esse campo. XML é atualmente muito usada e se tornou a base dos protocolos para *web services*.
XP
Abreviação comumente usada para Extreme Programming.
Z
Uma linguagem de especificação formal baseada em modelos desenvolvida na Universidade de Oxford, na Inglaterra.

Índice remissivo

D

Q

R

Ian Sommerville

ENGENHARIA DE SOFTWARE

9ª edição

No intuito de atender às necessidades de alunos e professores dos cursos de ciência da computação, engenharia da computação e sistema de informação, esta nona edição de *Engenharia de software* teve seu conteúdo reestruturado e totalmente atualizado.

A obra conta agora com novos capítulos que focalizam o desenvolvimento ágil de software e os sistemas embarcados, além de trazer novas abordagens sobre engenharia dirigida a modelos e desenvolvimento *open source*.

Essas novidades somadas à manutenção dos pontos fortes que consagraram a obra, como a clareza de conteúdo e a didática dinâmica, fazem dela um excelente guia para as mais diversas aplicações, independentemente das ferramentas ou processos de software disponíveis ou propostos.

www.pearson.com.br/sommerville
O site de apoio do livro oferece: para professores, sugestões de como usar o livro em sala de aula, banco de exercícios, manual de soluções (todos em inglês) e apresentações em PowerPoint; para estudantes, estudos de caso e quatro capítulos adicionais.

www.pearson.com.br

APRENDENDO SEMPRE

PEARSON